EASy Variability Instantiation Language: Language Specification

Version 0.97

(corresponds to VIL bundle versions 1.1.0, VTL bundle version 1.1.0 and rt-VIL bundle version 0.0.3)

H. Eichelberger, K. Schmid

Software Systems Engineering (SSE)

University of Hildesheim

31141 Hildesheim

Germany

Abstract

Deriving configured products in product line settings requires turning variabilities into artifacts such as source code, configuration files, etc. Frequently, this is done by domain-specific plugins into the product line tool, often called instantiator, which requires deeper-level programming knowledge about the underlying tool infrastructure. In this document we provide a novel approach for variability implementation. We focus on how to implement selected customization and configuration options in a generic way focusing on the specification of the instantiation process. This enables domain engineers to define their specific instantiation process in a declarative way without the need for implementation of specific tool components such as instantiators.

In this document we define and explain the concepts of the Variability Instantiation Language (VIL) for specifying how customization and configuration options can be turned into (instantiated) artifacts.

Version

0.5	15. June 2013	first version derived from INDENICA D2.2.2
0.6	8. August 2013	revised concepts
0.7	26. September 2013	revisions based on actual implementation
0.8	21. October 2013	Jars and zips added, clarifications for 'map', RHSMATCH and further built-in operations, qualified names, script parameter sequence refined, pattern path vs. artifact creation clarified, ITER clarified, further XML operations
0.85	30. October 2013	Starting the "How to ...?" section, classloader registration for Java Extensions in VTL
0.86	14. November 2013	Additional XML operations such as constructors, SCRIPTDIR VIL variable
0.87	12. December 2013	Refined hierarchical import path, named-base access to VTL scripts (in addition to file-path based access), final separator expression in for-loop, toJavaPath
0.87	19. December 2013	Java artifacts
0.88	10. February 2014	Project predecessors and further project operations, instantiation via scripts in other projects (instantiate command)
0.89	19. March 2014	Revision of automatic conversions (most specific one is applied).
0.90	04. April 2014	Versions for VTL in VIL scripts.
0.91	21. April 2014	Implementation status, detail fixes, renamePackages in Java Artifact
0.92	23. May 2014 – 02. July 2014	Qualified types, @advice introduces IVML types (Section 3.3.11), null (Section 3.3.9), syntactic additions for the iterated execution in VIL (Section 3.1.9.6), explicit iterator variables instead of ITER (Section 3.3.10)
0.93	03. July 2014 – 12. September 2014	Clarification of external executions in VIL, clarification of load properties, clarification of default extensions and their bundles (document structure changed), EnumValues, "AnyType" renamed to "Any" to avoid conflicts with "Type", random numbers, storing the configuration. Initial additions for instantiating variabilities at runtime (DSPL). Additional functions in Sequence and String. Conversion to sequence for DecisionVariable.
0.94	13. September 2014 – 02. February 2015	Append for sequences, including for sets. Versioned import clarified. Version constraints and Internal version type. Defer declaration removed (cleanup). Rule separating colon is now optional (for convenience). If-statement for VIL. Default extensions for VTL, making the default Random extension transparent. Integer-based sequence creation. `isConfigured` operation for decision variables. Further operations for map. Unmodifiable collections. ,`map` operation for sequences. Support for nested types. Maven instantiator for VIL. Creation of XML root nodes and how-to on XML. Split operation for String. JavaPath operations.
0.95		Inclusion of rt-VIL, optional explicit return type for rules ("functions"), setText operation for text representations, clarifications for XMLFileArtifact including a new how-to section, conversions for XMLFileArtifact. Maven

		instantiator integration improvements, rule refinements. Field notation for @advice, IVML mapping changed, enabling runtime variable changes, separation of the Java default extension into ANT and AspectJ extension. Generic sort function for sequences as well as revert function. Println for debug.
0.96	30. June 2015	Clarification on “Generics”, e.g. Set<Type> changed to setOf(Type), clarification on [] operator.
0.97	10. August 2015	collect and apply for collections, deleteStatement for Java. FROM/TO instead of RHS/LHS, attributes renamed to annotations according to IVML, asSet/asSequence added as alias operations. protected sub-templates and line end formatting for VTL. Clarification of global settings and deleting java calls, disallow cyclic imports. Fail statement for rt-VIL.

Document Properties

The spell checking language for this document is set to UK English.

Acknowledgements

This work was partially funded by the

- European Commission in the 7th framework programme through the INDENICA project (grant 257483).
- European Commission in the 7th framework programme through the QualiMaster project (grant 619525).
- German Ministry of Economic Affairs and Energy through the ScaleLog project (grant KF2912401SS).

We would like to thank Aike Sass (Java instantiator, VIL translation-time content expressions, serializer), Bartu Dernek (content assist) and Sebastian Bender (clarifications, ANT and Make instantiator) for their contributions.

Table of Contents

Table of Figures

1 Introduction

This document specifies the Variability Instantiation Language (VIL) in terms of the most current version of the language.

VIL consists of three languages: a build process description language, a template language and a runtime instantiation language, actually an extension of the build process language. The focus of VIL is on instantiating a whole product line, while the template language focuses on the instantiation and creation of individual artifacts. All VIL languages are based on an explicit and extensible artifact model as well as a tight integration with the Integrated Variability Modelling Language IVML [3].

The remainder of this language specification is structured as follows: in Section 2, we briefly introduce the VIL approach. In Section 3, we define the syntax and semantics of the aforementioned VIL languages, their common expression language as well as the underlying type system (including the artifact model and the IVML integration). In Section 4 we provide answers to frequently asked questions in terms of a how-to collection. In Section 5 we discuss the implementation status, in particular known deviations from this language specification. Finally, in Section 6, we provide the grammars of the VIL languages as a reference.

2 The Approach

In this section, we describe the concepts of the Variability Instantiation Language (VIL). VIL is designed to realize the instantiation of artifacts in a generic way, i.e., using a specification-based approach instead of relying on domain- or product-line-specific implemented instantiation mechanisms. A more detailed discussion of the approach idea and its benefits for product line engineering can be found in [5].

The VIL is more than a single language. It consists of three languages and requires the understanding of additional core concepts:

- **Artifact meta-model:** Everything that can be instantiated (transformed or generated) is regarded as an artifact. The VIL approach relies on an artifact meta-model as its foundation. The artifact meta-model (or often artifact model for short) describes what operations can be performed on certain types of artifacts, such as Java source code, Java byte code, XML files but also components (for runtime variabilities), or elements of the file system such as files or folders. Production strategies are operations on the types of the input and output artifacts using the capabilities of the assets for specifying the instantiation.
- **VIL template language** is used to instantiate a certain type of target artifact in a reusable way. Basically, the VIL template language covers generation as well as transformation-based production strategies.
- **Blackbox instantiators:** In some situations it might be difficult, inconvenient, or even impossible to describe a production strategy using the VIL template language. One example is the Cocktail instantiator discussed in Deliverable D2.2.2 [5] as it mainly modifies Java bytecode and, thus, it is easier to realize (at least some part of it) in an usual programming language such as Java, i.e., from the point of view of VIL as a black box. Another example is a programming language compiler or a linker, which should not be re-developed using VIL but simply reused. In case of legacy product lines, an existing instantiator may be called or wrapped into a VIL extension.
- **VIL control language:** This is the main part of VIL as it binds all other pieces together. This is used to define individual production strategies, i.e., to relate artifacts and instantiation mechanisms, to combine production strategies in terms of rules and to specify the execution of the rules. Basically, it is a rule-based programming language as a foundation for describing product line instantiation processes. We will call this language for short VIL.

VIL and its sublanguages are tightly integrated with IVML, i.e., IVML identifiers, configuration values, and types can be directly used in VIL. From a more general point of view, VIL and its sublanguages rely on existing, practically proven concepts such as build rules or template languages to avoid reinventing the wheel. However, existing concepts as well as related tooling does not provide the full support for variability instantiation as we experienced in our analysis of related technologies. Thus, we reuse and extend existing concepts to apply it to variability realization and created the VIL as a completely new language along with a novel implementation.

3 The VIL Languages

In this section, we will describe the two (sub) languages of VIL, i.e., the VIL (control) language and the VIL template language (VTL), as well as their main concepts. While VIL aims at defining the overall instantiation process for a software product line VTL specifically aims at supporting the generation and instantiation of textual artifacts. We describe VIL in Section 3.1 and VTL in Section 3.2. Due to the nature of both (sub) languages, they share a common type system as well as a common expression language. For working with different kinds of artifacts and variabilities, VIL provides an extensible type language. We will describe the expression language in Section 3.3, the type library in Section 3.4 and default extensions for specific programming languages in Section 3.5. For performing reconfigurations at runtime, VIL provides a specific set of concepts (we call this extending set runtime VIL, rt-VIL for short). We will describe rt-VIL in Section 3.6.

The **extensible VIL type system** is the foundation for both VIL sub languages. The type system consists of basic types such as Integer or Boolean, configuration-related types realizing the integration with an IVML [3] variability model, artifact-related types implementing the artifact (meta) model, implicit types representing instantiators and derived types such as containers. In particular, the type system is extensible, i.e., additional or refining artifact types or instantiators can easily be added (in terms of Java classes). If compared with an object-oriented language, the artifact types can be considered as classes, the operations as methods, individual artifacts as instances and the execution of artifact operations as method calls. However, the instantiators can be more aptly compared to transformation rules as they are first of all rule-based and second operate on the artifact model, but are themselves not part of it. The **common VIL expression language** represents a wide range of expressions from simple calculations over artifact operation and instantiator calls up to rather complex composite expressions. The expression language relies on the operations and operators provided by the VIL type system.

The two VIL languages are realized on top of the common expression language. Both languages follow a textual approach to the specification of artifact and product instantiation and support batch processing. Our definition of the syntax of VIL draws upon typical concepts used in programming languages, in particular Java, build languages such as *make*, template languages such as *xtend* as well as expressions inspired by IVML and the Object Constraint Language (OCL) [6]. We adapt these concepts as needed to provide additional operations required in variability instantiation, such as the integration with a variability model or an explicit artifact model.

We will use the following styles and elements throughout this section to illustrate the concepts of the IVML:

- The syntax as well as the examples will be illustrated in `Courier New`.
- **`Keywords`** will be highlighted using bold font.

- `Elements and expressions` that will be substituted by concrete values, identifiers, etc. will be highlighted using italics font.
- We will denote optional parts of syntax descriptions by *[...]*.
- Identifiers will be used to define names for modelling elements that allow the clear identification of these elements. We will define identifiers following the conventions typically used in programming languages. Identifiers may consist of any combination of letters and numbers, while the first character must not be a number.
- Statements will be separated using semicolon "`;`" (most other language concepts may optionally be ended by a semicolon).
- Different types of brackets will be used to indicate lists "`()`", sets "`{}`", etc. This is closely related to the Java programming language.
- We will indicate comments using "`//`" and "`/* ... */`" (cf. Java).

We will use the following structure to describe the different concepts:

- **Syntax:** this is the syntax of a concept. We will use this syntax to illustrate the valid definition of elements as well as their combination.
- **Description of syntax:** provides the description of the syntax and the associated semantics. We will describe each element, the semantics and their interaction with other elements in the model.
- **Example:** the concrete use of the abstract concepts is illustrated in a (simple) example.

In Section 3.1, we will describe the specific concepts of the VIL which is responsible for specifying the overall variability instantiation process of an entire (hierarchical) product line. In Section 3.2, we will describe the concepts of the VIL template language, which provides the means to describe the instantiation of a single (textual) artifact. Basic concepts of the VIL build and the VIL template language are rather similar (also to IVML) in order to simplify learning and application of these languages. In Section 3.3, we will detail the common expression language which is part of both, the VIL build and the VIL template language. In particular, we will detail the type system, i.e., the built-in types, their individual operations and the default instantiators that are part of the VIL implementation. In Section 3.4 we discuss the built-in VIL types and operations, i.e., the extensible library of types and operations that can be used to instantiate variability. In addition to the built-in operations, there are several default extensions of VIL, e.g., for Java artifacts. Extensions are optional and may or may not be installed depending on the actual product line setting. In Section 3.5, we discuss the default extensions for VIL.

As of version 0.95, we extended VIL for runtime instantiation, reconfiguration and adaptation through an extension of VIL, the Runtime Variability Instantiation Language (rt-VIL). In Section 3.6 we introduce the additional language concepts of rt-VIL and describe their semantics. In Section 3.7, we discuss the built-in operations for rt-VIL.

3.1 Variability Instantiation Language

In this section, we describe the concepts and language elements of the VIL (control) language (for short VIL) in detail. This language aims at specifying the variability instantiation process of a whole (hierarchical) product line (as opposed to the instantiation of a specific artifact type covered by the VIL template language).

However, VIL focuses on the implementation and instantiation of variabilities rather than on the entire build process of a whole system. Thus, VIL is intended as an extension to existing build languages, i.e., it shall be integrated with those languages rather than replacing them.

3.1.1 Reserved Keywords

In VIL, the following keywords are reserved and must not be used as identifiers. Please note that this set of reserved keywords is complemented by the keywords of the common VIL expression language given in Section 3.3.1.

- `@advice`
- `const`
- `else`
- `exclude`
- `extends`
- `execute`
- `if`
- `import`
- `instantiate`
- `join`
- `load`
- `properties`
- `protected`
- `requireVTL`
- `typedef`
- `version`
- `vilScript`
- `with`

3.1.2 Scripts

In VIL, a script (`vilScript`) is the top-level element. This element is mandatory as it identifies the production strategies to be applied to derive an instantiated product. The definition of a script requires a name, as a basis for referring among VIL scripts and a parameter list specifying the expected information from the execution environment such as the actual configuration or the projects to work on. In order to realize the necessary capabilities required for implementing the hierarchical product line capabilities of the EASy producer tool, at least the source project to instantiate from, the target project to be instantiated and the actual variability configuration must be passed to a VIL script.

Basically, VIL may refer to all visible configuration settings in a variability configuration. In traditional product line settings, VIL will refer to the actual values of frozen decision variables (and their underlying structure). In Dynamic Product Line Settings (DSPL) [10, 11], VIL will refer to all variables including unfrozen ones.

In order to make this integration with the variability model explicit, these decision variables may be directly referred in VIL by their qualified IVML name. As IVML configurations may be partial or even dynamically composed, the actual structure of a variability model is not necessarily known at the point in time when the VIL script is specified. Thus, the validity of qualified IVML identifiers can only be determined at execution time of the VIL script when also the actual configuration is known. This may complicate the development of VIL scripts as actually unknown identifiers will at least lead to a warning. To support the domain engineer in specifying valid scripts, VIL provides the `advice` annotation specifying the name for the IVML models used in the VIL scripts. Qualified names resolvable via the `advice` annotation do not lead to warnings in the VIL editor. Further, all types in IVML models (in case of overlaps in terms of their qualified names) are made available. On these types, each declared decision variable is available through an operation named according to the decision variable in order to ease the access. As an explicit version number may be stated for VIL scripts (akin to IMVL models), also advices and model imports may be version-constrained.

Optionally, a VIL script may extend another VIL script, i.e., reusing and extending production strategies by overriding (akin to object-oriented languages).

Syntax:

```
//imports
@advice(ivmlName)
vilScript name (parameterList) [extends name1] {
 //optional version specification
 //loading of variable values from an external source
 //variable definitions
 //rule declarations
}
```

Description of syntax: the definition of a VIL script consists of the following elements:

- First, all referenced scripts must be imported. We will detail the import syntax in Section 3.1.4.
- Optional advices declaring the underlying variability models.
- The keyword `vilScript` defines that the identifier *name* is defined as a new script with contained production strategies.
- The parameter list denotes the arguments to be passed to a VIL script for execution. When executed by EASy, at least the source project(s), the target project, and the variability configuration must be passed in. Source and target project may be identical in case of (traditional) in-place instantiation. In case of multiple source (predecessor) projects, the source

parameter shall be given as a collection. During processing, subsequent predecessors may be accessed via the `Project` type. However, further named parameters may be given upon an explicit invocation from an external call, e.g., an integration with a build language such as ANT or Maven. If given as first parameters in sequence project-configuration-project, the parameters will be bound independently of their name and, thus, the parameters can be named arbitrarily. Otherwise (due to the option of named parameters), the sequence of the parameters may be arbitrary but the parameters must exactly be named `source`, `config` and `target`.

- A VIL script may optionally extend an existing (imported) VIL script. This is expressed by **`extends`** *name$_1$*, whereby *name$_1$* denotes the name of the script that shall be extended.
- Production strategies are described within the curly brackets.

Example:

```
@advice(YMS)
vilScript YMSBuild(Project source, Configuration config,
       Project target){
  /* Go on with the production strategies for YMS here */
}
```

Please note that the types shown above such as `Project` or `Configuration` will be explained in detail in the next section. Further, a VIL script for multi-product lines may require a container of projects (see Section 3.1.5.4), while single project parameter is sufficient for a traditional product line build.

A script defines the following two implicit variables:

- `OTHERPROJECTS` is a collection variable which contains the artifacts created by the execution of VIL scripts in other projects.
- `SCRIPTDIR` is a String variable containing the location of this script in the file system.

3.1.3 Version

Akin to IMVL, VIL build specifications may optionally be tagged with an explicit version number in order to support product line evolution. Evolution of software may yield updates to projects, IVML models and VIL scripts so that scripts of different versions may exist and need to be clearly distinguished.

Syntax:

```
// Declaration of the version of a VIL script.
version vNumber.Number;
```

Description of Syntax: A version statement consists of the following elements:

- The **`version`** keyword indicates a version declaration. At maximum one version declaration may be given in a VIL build file at the very first position within a VIL script.
- ***`v`****`Number.Number`* defines the actual version of the project (here only two parts prefixed by a "v"). At least one number must be given and no restriction holds on the amount of sub-version numbers.
- A version statement ends with a semicolon.

Example:

```
vilScript YMSBuild(Project source, Configuration config,
     Project target){
     version v0.1.4;
     ...
}
```

3.1.4 Imports

The production strategies for a variability instantiation build process may be defined in a single VIL script or may be reused from other (existing) scripts. Therefore, VIL scripts may be imported. Cyclic imports are not allowed and shall cause an error. In order to support also the evolution of product line build specifications, VIL allows the specification of version-restricted imports. Imports make the production strategies defined in the specified build file accessible to the importing script.

Syntax:

```
// Unconstrained and constrained imports.
import name;
import name with expression;
```

Description of Syntax: An import of a scripts consists of the following elements:

- Importing a VIL script starts with the keyword **`import`**. Multiple imports may be given in a VIL build file directly at the beginning of the script file.
- *`name`* (given in terms of a VIL identifier) refers to the name of the script to be imported. However, multiple scripts with identical names and versions may exist in a file system, in particular in hierarchical product lines. Thus, imports are determined according to the following **hierarchical import convention**, i.e., starting at the (file) location of the importing script (giving precedence to imports in the same file) the following locations are considered in the given sequence: The same directory, then contained directories (closest directories are preferred) and finally containing directories (also here closest directories are preferred). In addition, sibling folders of the folder containing the importing model and predecessor

projects are considered[3]. Similar to Java class paths, additional script paths may be considered in addition to the immediate file hierarchy.

- An optional restriction of the import in terms of an expression. This is indicated by the keyword **`with`**[4]. followed by a parenthesis containing the restrictions. A restriction is stated using the implicit variable **`version`** denoting the version of the import an operation supported by the version type (==, >, <, >=, <=, see Section 3.4.4) and a version number. More complex constraints can be stated using Boolean operations. In case of multiple matching versions, the model with the highest version number is selected by default.
- An import statement ends with a semicolon.

Example:

```
vilScript YMSBuild(Project source, Configuration config,
      Project target){
      version v0.1.4;
      import generics with (version >= v1.12);
}
```

VTL templates may not be imported but restricted with respect to their version. The syntax looks similar to imports, but the syntax is adjusted to the actual way VTL templates are called from VIL. Using the syntax below, VTL scripts with name "name" are restricted to the given version akin to the import of VIL scripts.

```
restrictVTL "name" with expression;
```

3.1.5 Types

Basically, VIL is a statically typed language with partially postponed type checking at runtime as we will detail below. Thus, VIL provides a set of formal types to be used in variable declarations or parameter lists. We distinguish between basic types, configuration types, artifact types, and container types.

3.1.5.1 Basic Types

The basic types in VIL correspond to the basic types of IVML, i.e., Boolean (`Boolean`), integer (`Integer`), real (`Real`) and string (`String`) with their usual meaning.

Boolean constants are given in terms of the keywords true and false. Integer constants are stated as usual numbers not containing a "." or an exponential notation. Real constants must contain the floating-point separator "." or may be

[3] Actually, EasY-Producer stores the imported parent product line models in individual subfolders (starting with a "."), i.e. possibly sibling folders of a model.

[4] Previous versions required parenthesis and supported only a rather limited set of expressions. Since version 0.93, arbitrary expressions can be used here, but currently we support only version expressions.

given in exponential notation. Strings are either given in quotes or in apostrophs and may contain the usual escape sequences including those for line ends, quotes and apostrophs.[5]

3.1.5.2 Configuration Types

A configuration type denotes the representation of IVML configuration elements in VIL. However, due to the nature of VIL, we need only access to the configuration and the structure of an IVML model rather than to all modelling capabilities. Thus, VIL provides a specific set of built-in configuration types. The actual instance of a configuration is passed into a VIL script in terms of a script parameter. Configuration types cannot be directly created in a VIL script and must not modify the underlying IVML model.

The entry point to a configuration in terms of an IVML model is the type `Configuration`. It provides access to all decision variables and annotations, depending on the product line settings frozen variables only or all variables. In particular, `Configuration` allows creating projections of a given configuration in order to simplify further processing. Further, it provides access to IVML type declarations such as compounds or enumerations and their value. This is represented by the `IvmlElement` and its subtypes. An `IvmlElement` represents IVML concepts in a generic way and provides access to its (qualified) name, its (qualified) type name and the configured value. Specific subtypes of `IvmlElement` are `DecisionVariable`, `Annotation` and `IvmlDeclaration`, each providing more specific operations as we will discuss in detail in Section 3.4.4.

3.1.5.3 Artifact Types

Artifact types represent the different categories of artifacts used in the artifact model. Some artifact types are built-in and part of the VIL implementation, while further types can be defined in terms of an extension of the artifact model. In this section, we will discuss only the predefined types. Please refer to the EASy developers guide on how to define more specific artifact types (as well as how to integrate instantiators implemented in a programming language).

The type `Project` is a mapping of a physical project (Eclipse) into VIL and provides related operations such as mapping paths between the source and the target project for instantiation.

A `Path` is a predefined type of the VIL artifact model although it is not an artifact by itself. A `Path` represents a relative file system path and may possibly contain wildcards. A path is specified in terms of a `String` in VIL and is automatically converted into a `Path` or an artifact instance depending on the actual use. In more detail, paths are specified according to the ANT [9] conventions, i.e., using the slash as path separator and wildcards for patterns. The following wildcards are supported: `?` for a single character (excluding the path separator), `*` for multiple characters (excluding the path separator) and `**` for (sub) path matches.

[5] Strings delimited by quotes may contain apostrophs, strings delimited by apostrophs may contain quotes.

`Artifact`[6] is the most common artifact type and root of the VIL artifact hierarchy. The predefined `Artifact`s have also predefined methods. For example, they allow to delete the artifact (if possible at all), or to obtain access to its plain textual or binary representation. VIL provides a set of built-in artifact types such as `FileArtifact` and `FolderArtifact` which are both `FileSystemArtifact`s. Further, VIL provides more specific artifact types such as the `VtlFileArtifact` representing VIL template files (see Section 3.4.6) or the `XmlFileArtifact` representing parsed XML files with a substructure of specialized fragment artifacts such as `XmlElement` or `XmlAttribute`. Please note that artifact instances are assigned in a polymorphic way, i.e., while a `FileArtifact` may be specified as type in a VIL script, it may actually contain a more specific type. However, pattern paths, i.e., paths containing wildcards, will not be turned into artifact instances.

3.1.5.4 Container Types

VIL provides three container types, sequences (keyword `sequenceOf`), sets (keyword `setOf`) and associative containers (keyword `mapOf`). Container types are generic with respect to their content type(s) and, similarly to IVML, the content type must be stated explicitly, such as `sequenceOf(Integer)` or `setOf(DecisionVariable, FileArtifact)`.

Sequences may contain an arbitrary number of elements of a given element type (including duplicates), while sets are similar to sequences, but do not support duplicate elements. In sequences, elements can be accessed by their position in the container using an index (`[`*`index`*`]`). In VIL, indexes start at zero and run until the number of elements in the container minus one (as in Java and many other languages). Collections typically occur as results of operations, rule, or instantiator executions. In addition, they can be explicitly initialized using type-compatible expressions of the appropriate dimension as shown follows

```
sequenceOf(Integer) someNumbers = {1, 2, 3, 4, 5};
setOf(Integer, Integer) somePairs = {{1, 2}, {3, 4}};
```

A `Map` represents an associative container in VIL, i.e., a container which relates keys to associated values. In particular, it allows retrieving the value assigned to a key via the `get` operation and the `[]`-Operator (`[`*`key`*`]`). Basically, associative containers are intended to simplify the translation of IVML-identifiers to implementation-specific identifiers in individual artifacts. Therefore, VIL associative containers can be explicitly initialized in terms of key-value-pairs using type-compatible expressions

```
mapOf(String, String) idTranslation
  = {{"nrOfProcessors", "procCnt"}, {"nrOfNodes", "nodeCnt"}};
```

VIL supports a set of operations specific for container types, e.g., excluding, projecting, or collecting elements in a container, etc. We will introduce the full set of operations in Section 3.4.3.

[6] We adopted US English in the implementation of VIL.

3.1.6 Typedefs

Generic types such as the collections can lead to rather complex but inconvenient type names. A typical example is

```
mapOf(String, mapOf(String, String)) data;
```

In order to simplify the use of such types and to increase type safety, we allow the definition of type aliases akin to C or the unconstraint typedefs in IVML.

Syntax:

```
typedef name Type;
```

Description of Syntax: The declaration of typedefs consists of the following elements:

- The keyword **`typedef`** indicates the declaration of an alias type.
- The identifier *`name`* denotes the name of the new (alias) type.
- The *`Type`* is an existing type expression used to define the alias.

Type aliases can be used wherever the original type can be used.

Example:

```
typedef DataType mapOf(String, mapOf(String, String));
```

Then the expression shown above becomes

```
DataType data;
```

3.1.7 Variables

A variable provides name-based access to a value of a certain type (see Section 3.1.5), similar to variables in programming languages.

In VIL, the value of a variable can be modified at any time (in contrast to build languages such as ANT [9] where a value of a property can be set only once). In addition, a variable may be declared to be constant so that a value can be set only once and not be modified afterwards. Variables may be of global scope, i.e., directly defined within a VIL script or they may be local (within rules, see Section 3.1.6).

Syntax:

```
// Declaration of a variable.
Type variableName1;
Type variableName2 = value;
const Type constantName = value;
```

Description of Syntax: The declaration of variables consists of the following elements:

- The ***`Type`*** defines the type of the variable being declared.
- The identifiers *`variableName`*$_1$, *`variableName`*$_2$ and *`constantName`* are the names of the declared variable or constant, respectively.

- The optional keyword `const` indicates that a variable can be defined only once and the value must not be redefined.
- A variable may optionally be initialized by a value or an expression, which evaluates to a value of the given type.
- Variable declarations end by a semicolon.

Parameters of VIL scripts are declared akin to variables, but without an initial value.

Example:

```
Integer numberOfCompilerProcesses = 4;
sequenceOf(Project) sources;
sequenceOf(Project, DecisionVariable) mapping;
```

Variables may be referred in Strings such as path patterns. A variable reference is stated as `$variableName`. Even entire VIL expressions (see Section 3.3) including variables may be given in Strings in the form `${expression}`. When applying the respective String, variable, and expression references are substituted with their actual value.

3.1.8 Externally Defined Values of Global Variables

Global variables or constants are defined as part of a VIL script. The value of a global variable or constant may be specified by an external source, e.g., to customize the VIL script according to the build environment (similar to properties in ANT [9]). For externally defined values of variables, initial values are not needed, in particular also not for constants. Externally specified values are subject to automated type conversion and variable reference or VIL variable (not expression) substitution based on the VIL script arguments. Multiple external property files are processed in sequence so that the variable values defined by external files listed before are overwritten (accidental constant redefinition will lead to an execution error).

Syntax:

```
// Loading the values of global variables
load properties "path";
```

Description of Syntax: Loading values from an external file consists of the following elements:

- The keywords ***load properties*** indicates that the values of global variables shall be loaded from an external file. Multiple load statements may be given in a VIL script directly after the version statement.
- The path points to the file containing the initial values of the variables. Relative paths are interpreted relative to the target project of the VIL script, i.e., in case of multiple instantiations all properties must be set in the (partial) target project. Also absolute paths may be given, in particular using variable references as described in Section 3.1.6. The file must be given in Java properties format, i.e., each line specifies the value of a specific variable in the following form
  `variableName = value`
  In addition to the given file, VIL tries to load operating system specific files afterwards overriding previous settings, i.e., considering the given path as basis, augmenting it with an operating system specific infix (`win` for Windows, `macos` for MacOs and `unix` for all other Unix-like systems) before the filename extension. For example, in addition to `my.properties` on Windows, VIL also tries to load also `my.win.poperties` (no error will occur if operating specific properties are not given). Please note that the variables described in the properties file must be defined as global variables (outside any Rule) in the VIL script.
- Loading values from an external file ends by a semicolon.

Example:

```
load properties "globalVariables.properties";
Integer variableName;
```

Fills the global variables in the Script with values defined in the given properties file. Assuming that `variableName = 1` is stated in the properties file, the value of the variable `variableName` will be `1` after loading the properties.

3.1.9 Rules

Build rules are used in VIL to specify individual production strategies, reusable build steps to be used within production strategies or, as the main entry point into the build process. Akin to *make* [8], VIL-rules may have preconditions, which must be fulfilled in order to enable the rule. However, VIL-rules may also explicitly define postconditions, which guard the result of the rule execution.

VIL rules may have **parameters** in order to parameterize the specified variability instantiation. These parameters must either be bound by the calling rule or, in case of the main entry rule, by the VIL script itself.

Preconditions may be given in terms of path-patterns, an individual artifact, an artifact collection or rule calls. While an arbitrary number of rule calls may be given as precondition, at most one path pattern, artifact or artifact collection may be given as first precondition[7]. If the preconditions of a rule are not fulfilled, the rule is not considered for evaluation, i.e., it also does not fail.

- A path pattern follows the (pattern) rules of ANT path specifications already described in Section 3.1.5.2. For example, `"$target/bin/**/*.class"` requires the existence of at least one Java bytecode file in the `bin` folder of `$target` (assuming that `$target` refers to the target project). Used as a rule precondition, a path pattern requires that the matching file artifacts exist and are up-to-date (akin to Make rule preconditions [8] but with extended pattern matching capabilities).
- An artifact (collection) is given in terms of a variable or a VIL expression evaluating to exactly one artifact (collection) instance. In a precondition, the denoted artifact(s) must exist and be up-to-date.
- A rule call (rule name with argument list) represents an explicit rule dependency and must be executed successfully in case that the preconditions of the stated rule are valid. The execution results of a rule call become available as an implicit variable in the rule body under the name of the called rule.
- A Boolean expression based on parameters, global variables or a (function) rule calls.

The optional **rule postcondition** is given in terms of a path pattern, an individual artifact, an artifact collection or a Boolean expression. Postconditions are evaluated if the preconditions are met and the body of the rule is executed successfully. A rule completes successfully, if also the (optional) postcondition is met.

Rules may explicitly depend on each other in terms of the rule calls described above. Further, **implicit rule dependencies** are expressed via the first (non-rule call) pre- or postcondition (akin to *make* rules [8]). If a path matching precondition for rule r_0 is not fulfilled, the VIL execution environment will aim at fulfilling the precondition by (recursively) searching for rules r_i with a postcondition indicating that the successful execution rule r_i contributes to the unmet precondition of rule r_0. Ultimately, the possibly contributing rules r_i are executed (including their implicit rule dependencies) and the precondition of rule r_0 is checked again, and, on fulfilment, also r_0 is executed. If finally the precondition of r_0 is not fulfilled, r_0 is not considered for execution.

[7] Future work on VIL may relax this condition and even extend the current file path notation to more generic artifact model path expressions also involving fragment artefacts etc.

The rule body specifies the individual steps to be executed if the preconditions are met. A rule body may contain variable declarations, (assignment) expressions, explicit rule calls (not relevant as preconditions), instantiator calls, execution of system commands, or iterated execution of the previous elements. We will first describe the syntax of rules and describe then the individual statements available for specifying rule bodies.

If no path matching precondition is given, the rule body is executed once. If a path matching precondition is present, one or multiple artifacts may match that precondition and for each of these artifacts a corresponding output artifact may be required by the postcondition (if specified). However, the related match conditions may directly be used in the rule body but then possibly lead to a (superfluous) re-instantiation of the related target artifacts. In order to avoid re-instantiation and to allow for optimizing the variability instantiation, VIL offers two ways of executing the rule body. These two ways are:

1) Passing the right hand side matches as an implicit collection variable called `FROM_MATCH` to the rule body. This is more appropriate for instantiations which may operate on multiple artifacts and consider dependencies among artifacts by themselves, such as a Java compiler.
2) Implicitly iterating over the matched pairs of left hand and right hand side artifacts. Then rule body is executed iteratively over all matching precondition artifacts. In order to address the actual artifact to be processed as well as its expected resulting artifact, the implicit variables[8] `TO` (in case of a matching precondition) and `FROM` (in case of a matching postcondition) will be made available to the loop body. This way is more appropriate for single artifact instantiations, such as calling a pre-processor or a C compiler.

By default, rules return implicitly their execution results consisting of two sequences,

- `result` containing the immediately modified artifacts by that particular rule.
- `allResults` containing the modified artifacts by all dependent rule calls.

In addition, rules may act as functions, i.e., they may declare a return type and the last expression in the rule must then comply to this return type.

Further, the artifacts modified by executed rules are be successively collected in the implicit global collection variable `OTHERPROJECTS`.

Syntax:

```
[protected] [Type] name (parameterList) = [[postcondition]
 : [preconditions]] {
 // TO/FROM/FROM_MATCH may be available in the body
 //variable declarations
 //rule, instantiator, artifact or system calls
 //iterated execution
}
```

[8] The old names RHS, LHS and RHSMATCH are still available, but deprecated. They will be removed in one of the next versions.

Description of Syntax: A rule declaration consists of the following elements:

- The optional keyword **`protected`** prevents that this rule is visible from outside so that such rules cannot be used as an entry point (for example, in ANT [9] this is expressed by a target name starting with the minus character) or from other scripts.
- An optional type (see second syntax form above) may be given in order to declare a "function". In this case, the last expression must return a value of the return type.
- The *`name`* allows identifying the rule for explicit rule calls or for script extension.
- The *`parameterList`* specifies explicit parameters which may be used as arguments for precondition rule calls as well as within the rule body. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Parameter must either be bound by the calling rule or, in case of the main entry rule (usually called `main`), by the VIL script itself (same parameter sequence and assignable values in both, the template and the main sub-template).
- The first three parts may be omitted in case of anonymous rules which are only executed due to implicit dependencies and not available to explicit rule calls.
- The optional postcondition specifies the expected outcome of the rule execution. A postcondition may be a path match, an artifact or an artifact collection. In case of a path match, the implicit variable FROM will be made available to the rule body.
- The optional preconditions specify whether the rule is considered for execution. The first precondition (made available as implicit variable TO to the rule body) may be a path match, an artifact or an artifact collection. The following preconditions may be explicit rule calls. The execution results of the preconditions will be made available to the rule body in terms of implicit variables with names of the called rules and the rule return type described above. If neither pre- nor postcondition are given, also the separating colon can be omitted.
- The rule body is specified within the following curly brackets.

Example:

```
produceGenericCopy(FileArtifact x, FileArtifact y) = y : x
{
    x.copy(y);
}

compileGoal() = "$target/bin/*.class" : "$source/*.java"
{
    javac(FROM, TO);
}
Integer helper(Integer param) =
{
    param + 100;
}
```

The rule body specifies the individual steps to be performed in order to fulfil the rule postcondition (if stated). A rule body may contain variable declarations, (assignment) expressions, explicit rule calls, instantiator calls, execution of system commands or iterated execution of these elements. The statements (ended by a semicolon or a statement block) given in a rule body are executed in the given sequence. The `helper` rule defines a "function" returning a derived value, in particular to enable the reuse of complex expressions. We will discuss these individual elements in the following subsections.

3.1.9.1 Variable Declarations

A variable declaration within a rule body introduces a local variable shadowing rule parameters or global variables. This is in particular true for variables, which are defined within nested blocks. Basically, a variable declaration within a rule body follows the same syntax as global variable declarations discussed in Section 3.1.6.

3.1.9.2 Expressions

Expressions such as value calculations or execution of artifact operations may be used within a rule body as a guard expression or as a variable assignment. Please note that we will detail VIL expressions in Section 3.3, as the expression language is common to both, VIL and VTL.

- Guard expressions constrain the execution of the remaining statements in a rule body, i.e., the expression must be evaluated successfully in order to continue the execution of the rule.
- In a variable assignment, the expression on the right hand side of the assignment operator "=" must be evaluated successfully in order to assign the evaluation result of the right side to the variable specified on the left side.

As we will explain in Section 3.3 in more detail, expressions are evaluated lazily, i.e., expressions that are undefined, because a used value or operation call is evaluated to undefined are ignored and do not lead to the failing of containing rules.

3.1.9.3 Calls

A call leads to the execution of another rule, an instantiator or an artifact operation. We will discuss three types of calls in this section, as they are represented by the same syntax. However, the most extreme call of a (blackbox) instantiator, namely the execution of an operating system command (including operating system scripts) follows a slightly different syntax. We will discuss operating system commands in Section 3.1.9.4.

The syntax of rule calls, instantiators or artifact operations looks as follows:

```
operationName(argumentList)
```

whereby arguments are expressions separated by commas. Calls may return values of different type.

Rule Calls

An explicit rule call is stated in terms of the name of the rule and the arguments matching the parameter list of the target rule. A rule call leads to the execution of a rule defined in the same script, one of the extended scripts or an imported script. As rules with the same signature consisting of name and parameter list are shadowed by the extension, rules in extended scripts may explicitly be called by

```
super.operationName(argumentList)
```

Operations defined in imported scripts may be denoted by their qualified name as `operationName`, i.e., prepending the import path until the defining model.

Instantiator Calls

Basically, VIL aims at defining the production flow for instantiating generic artifacts for a software product line. In contrast, the VIL template language aims at specifying the individual actions to instantiate an individual (generic) artifact. Further instantiators may be given in terms of (wrapping) Java classes in order to make programming language compilers, linkers, or legacy instantiators available. Such instantiators may provide information about their execution, in particular the created artifacts.

In VIL, all these types of instantiators are mapped transparently to one kind of statement, the instantiator call:

```
operationName(argumentList)
```

Basically, an instantiator call looks similar to a rule call, i.e., a name with a parameter list, but it (typically) returns a collection of artifacts (or even nothing in case of wrapped blackbox instantiators). Instantiators may be rather generic (such as the built-in instantiator for the VIL template language) and may offer to pass an arbitrary number of arguments (e.g., those defined by a VIL template. Therefore, depending on the instantiator, named arguments (`parameterName = valueExpression`) may pass arbitrary VIL instances to an instantiator in a generic way.

We will detail the built-in instantiators in Section 3.4.7. Please refer to the developer documentation of EASy-Producer for obtaining information on how to realize an instantiator.

Artifact Operation Calls

Artifact operations provide information on an individual artifact, its fragments or even enable the manipulation of artifacts. Basically, an artifact operation is executed on a variable or expression, which evaluates to an artifact type. An artifact operation can be expressed (akin to IVML and OCL) in two different ways, using the artifact as first argument

```
operationName(artifact, argumentList)
```

or in object-oriented style

```
artifact.operationName(argumentList)
```

Basically, a String can be automatically converted into a Path or an `Artifact`. Similarly, a Path can be transparently converted into an `Artifact`. However, in some cases, also an explicit creation of an artifact of a certain type may be required. Typically, the individual artifact types support the following constructors

```
new ArtifactType(String)
```

for obtaining a specific artifact specified by its path. If the given *`ArtifactType`* does not support the actual format of the underlying artifact, e.g., a Java file shall be considered as an XML file, the resulting artifact is undefined and, due to lazy evaluation, subsequent expressions are ignored.

Artifacts are associated with creation rules. Basically, file artifacts (regardless of whether they physically exist or only the path is known) are polymorphically determined according to their file name extension, e.g., a file with extension `xml` is considered to be a `XmlFileArtifact`. However, pattern paths, i.e., paths containing wildcards, will not be turned into artifact instances. Further, content-specific rules may apply depending on the specific artifact type. If no such rule applies, a basic `FileArtifact` is created as the default fallback. Thus, the underlying mechanisms of the VIL artifact model will check whether the creation of that instance (regardless of whether the underlying file exists or not) is actually possible or not. If the creation fails, also the containing rule will fail.

Folders are created transparently, e.g., when the underlying file beyond a file artifact is created. The constructor

```
new ArtifactType()
```

allows to obtain a temporary file or folder artifact. Unless not renamed, this artifact will be automatically deleted after terminating the execution of the VIL script.

The modifications to a VIL artifact instance will automatically be synchronized with the underlying artifact upon the end of the lifetime of the related variable, e.g., when the execution of the containing scope of a local variable ends.

We will detail the built-in artifact operations in Section 3.4.6. Please refer to the developer documentation of EASy-Producer for obtaining information on how to realize own artifact types and related operations.

Operation Resolution

While determining the applicable rules, instantiators, or artifact operations, the VIL type system considers in the following sequence

(1) Exact match of argument types and parameter types.
(2) Assignment compatible argument types and parameters.
(3) Automatic conversions specified as part of the implementation of VIL types and artifacts, e.g., the implicit conversion of a `String` to a `Path` or a `String` to an `Artifact`. If multiple conversion operations may support the required conversion, the most specific one with respect to the type hierarchy of the source type will be applied. Details on the type system and the available conversions will be discussed in Section 3.3.

The operation types discussed in this section will be resolved according to the sequence below:

(1) Rule calls
(2) Instantiator calls
(3) Artifact, configuration type, and basic type operations

Further, the VIL runtime environment performs dynamic dispatch, i.e., the operation determined and bound at script parsing time will be reconsidered with respect to the actual types of parameters and the best matching operation will dynamically be determined (similar to dynamic dispatch in Xtend [2]). This avoids the need for explicit type checking or large alternative decision blocks.

3.1.9.4 Operating System Commands

VIL is also able to execute the most basic form of a blackbox instantiator, namely operating systems calls or scripts. However, the syntax for system calls differs slightly from the other call types discussed in Section 3.1.9.3 as operating system commands may require explicit path specifications.

```
execute identifier(argumentList)
```

whereby *`identifier`* must denote a variable which evaluates to a `String` or a `Path`.

The *`argumentList`* contains the arguments to be passed to the external command in terms of a comma separated list. Although it is in VIL syntactically correct to put multiple arguments into a single String, this leads typically to execution problems as the underlying infrastructure (Java) expects individual arguments and even inserts quotes to cope with whitespaces.

This enables that operating system calls can be composed at script execution time or determined using external values (see Section 3.1.8). However, the related command or script is executed, but the created artifacts are not tracked by the VIL execution environment.

3.1.9.5 Alternative Execution

In VIL, alternatives allow choosing among different ways of instantiating artifacts, i.e., upon evaluating a condition the appropriate alternative to execute is determined.

The (return) type of an alternative is either the common type of all alternatives or `Any`. The return value of an alternative is determined by the last statement executed in the alternative selected by the (condition) expression.

Syntax:

```
if (expression) ifStatement
if (expression) ifStatement else elseStatement
```

Description of Syntax: An alternative statement consists of the following elements:

- The keyword **`if`** indicates the beginning of an alternative statement.
- The *`expression`* given in parenthesis determines whether the if-part (condition is evaluated to true) or the else-part (condition is evaluated to false) is executed. If the *`expression`* cannot be evaluated, the alternatives will be evaluated.
- The *`ifStatement`* (or statement block enclosed in curly braces) is being executed when the *`expression`* is evaluated to true. A single statement must be terminated by a semicolon.
- The **`else`** part is optional, i.e., if else is used in an alternative, a following *`elseStatement`* is required. As usual, a dangling else is bound to the innermost alternative.
- The *`elseStatement`* (or statement block enclosed in curly braces) is executed if the *`expression`* is evaluated to false. Again, a single statement must be terminated by a semicolon.

Example:

```
if (config.variables().size() > 0) {
    // work on config
}
```

3.1.9.6 Iterated Execution

Finally, all statements available in a rule body may explicitly be executed in iterative fashion, e.g., to apply a sequence of instantiator calls explicitly to a container of artifacts. Therefore, VIL offers a dedicated loop statement. However, this expression called `map` in VIL is different from typical programming language loops as it collects the result of its execution in terms of modified artifacts (similar to a rule). The results produced by a map expression are determined by the last standalone expression in a map body. The evaluation results of these expressions are collected in a sequence of type of the expression. The results can be assigned to a variable. If there is no such expression, map will behave like a typical for-loop. However, due to its character as an expression, a map must be used within an expression and terminated by a semicolon.

Syntax:

```
map(names = expression) {
//variable declarations
//rule, instantiator, artifact or system calls
//iterated execution
};
```

Description of Syntax: An iterated execution consists of the following elements:

- The keyword **`map`** followed by parenthesis defining the iterator variables.
- The *`names`* denoting the names of the variables used by the map expression iterate. The number, type and the contents of the iterator variables are implicitly defined by the related `expression`. Optionally, in order to ease the learning for programmers, for each variable a type can be given which must comply with the respective inferred type from `expression`. Typically, the expression will lead to a container with one parameter type so that map will iterate over that collection using exactly one variable of the element type of the container. However, as we will discuss in Section 3.1.9.7, the `join` expression may return a multi-dimensional container, which then needs multiple iterator variables.
- The equality/assignment symbol separates the variables from the `expression`. In order to ease the application and the learning for programmers, alternatively a colon may be used here.
- The **`map`** consists of a block determining the statements to be executed in for an individual iteration. The *`names`* denoting the iterator variables shall be used within the block.

Example:

```
map(d = config.variables()) {
    // operate on the iterator variable d of type
    // DecisionVariable (see Section 3.4.5.6)
};
```

Below we depict the equivalent notation using a colon, an explicit type and the implicit conversion of a decision variable to a collection using its contained values. Please note that explicit types for the variables and the use of the colon are independent from each other.

```
map(DecisionVariable d : config) {
    // operate on the iterator variable d of type
    // DecisionVariable (see Section 3.4.5.6)
};
```

3.1.9.7 Join Expression

One specific expression in VIL is particularly intended to be used with the `map` iteration statement, namely the `join` operation. However, as `join` is an expression, it may be used as an usual expression, e.g., on the right hand side of a value assignment to a variable.

This operation is inspired by database joins, e.g., as usual in SQL. The `join` operation allows combining containers of different VIL types, in particular elements from the variability configuration with source or target artifacts. Depending on type of the specified expression types, the `join` operation returns a typed sequence containing the results.

Syntax:

join($name_1$**:**$expression_1$**,** $name_2$**:**$expression_2$**) with (***expression***)**

Description of Syntax: A VIL join expression consists of the following elements:

- The keyword **join** followed by one parenthesis defines the containers to be joined and the related iterator variables ($name_1$, $name_2$).
- $name_1$, $name_2$ denote variables used by the join expression iterate over the containers given in $expression_1$ and $expression_2$. Without further restriction, the result will be a collection of pairs on the types parameterized by the types of $expression_1$ and $expression_2$. The keyword `exclude` used before one of the names leads to a left- or right-sided join, thus restricting number of parameters of the resulting collection to one.
- The third `expression` specifies the join condition, i.e., an expression involving $name_1$ and $name_2$ to select the relevant results from the cross product of $name_1$ and $name_2$, and to effectively reduce the size of the result.

Example:

```
// work on those decisions and artifacts where a certain
// string composed from the decision name occurs in the
// artifact (and may be substituted by an instantiator)
map(d, a :
  join(d:config.variables(), a:"$source/src/**/*.java")
  with (a.text().matches("${" + d.name() + "}")) {
    // operate on decision variable d and
    // related artifact a
}
```

3.1.9.8 Instantiate Expression

In addition to the instantiation of individual projects, VIL is specifically intended to support the instantiation of hierarchical and multi product lines. Therefore, it is necessary to execute scripts in other projects, if required on projections of the configuration and with specific target project. Basically, applying already known VIL concepts, it is possible to explicitly call the main rule of predecessor projects statically, i.e., to import the predecessor projects and to call the respective main rule through its qualified name. However, the references to the predecessor projects would be static and not subject to possible variability. Thus, we introduce the instantiate expression, which allows executing a VIL rule in a project allowing to dynamically referring to VIL scripts in (other) projects. Per se, the instantiate expression does not imply a recursion over predecessor projects, but if the predecessor project realizes recursive instantiation, the instantiate expression will perform the recursion. Please note, that due to the dynamic resolution, rule calls via the instantiate expression imply a higher overhead.

Syntax:

```
instantiate name (argumentList) [with (expression)]
```

Description of Syntax: A VIL instantiate expression consists of the following elements:

- The keyword **instantiate** followed by either the name of a variable containing the project to instantiate or a string containing the qualified name of the rule to be executed.
- The argument list of the rule to be executed in parenthesis. Please note, that in particular also the source project (in form 1-3 the project the script is defined for), the (relevant part of the) configuration and the target project (typically the calling project) must be given (following the VIL parameter conventions). Further, optional named arguments may be given.
- Finally, the version specification of the model to be instantiated may be given.

Example:

```
// instantiate the predecessor projects into target
vilScript a (Project source, Configuration config,
  Project target) {
  map(Project p: source.predecessors()) {
    instantiate p (p, config, target);
  };
}
```

3.2 VIL Template Language

In this section, we describe the concepts and language elements of the VIL template language in detail. In contrast to VIL, which aims at specifying the instantiation of all artifacts of a product line, the VIL template language aims at specifying the instantiation of a single artifact.

3.2.1 Reserved Keywords

In the VIL template language, the following keywords are reserved and must not be used as identifiers. Please note that this set of reserved keywords is complemented by those of the common VIL expression language given in Section 3.3.1.

- **`@advice`**
- **`@format`**
- **`@indent`**
- **`const`**
- **`def`**
- **`default`**
- **`else`**
- **`extends`**
- **`extension`**
- **`for`**
- **`if`**
- **`import`**
- **`print`**
- **`protected`**
- **`switch`**
- **`typedef`**
- **`template`**
- **`version`**
- **`with`**

3.2.2 Template

The template (**`template`**) is the top-level structure in the VIL template language. This element is mandatory as it defines the frame for specifying how to instantiate a certain artifact. Please note that exactly one template must be given in a VIL template file.

The definition of a template requires a name, which acts for referring among VIL templates and a parameter list specifying the expected information from the calling VIL script such as the actual configuration and the target artifact (fragment). Please note that these two arguments must be provided to all VIL template scripts.

Basically, VTL may refer to all visible configuration settings in a variability configuration, more precisely to those actual values of decision variables (and their underlying structure), which are frozen. In order to make this integration explicit, these decision variables may be directly referenced in the VIL template language by their qualified IVML name. As IVML configurations may be partial or even dynamically composed, the actual structure of a variability model is not necessarily known at the point in time when the VIL script is specified. Thus, the validity of qualified IVML identifiers can only be determined at execution time of the VIL script when also the actual configuration is known. To support the domain engineer in specifying valid templates, also the VIL template language provides the **`advice`** annotation (see also Section 3.1.2).

Optionally, a VTL template may extend another VTL template, i.e., reusing and extending production strategies by overriding (akin to object-oriented languages).

VTL particularly aims at supporting generative and manipulative instantiation of generic artifacts. Therefore, the VTL provides capabilities for easily specifying and generating contents. However, as usual in software development, also VIL templates shall be formatted properly. In order to distinguish between intended formatting and whitespaces that shall not occur in the target artifact (fragment), the VIL template language is able to take the actual indentation into account (as specified in the **`indent`** annotation). Taking the formatting of the templates into account avoids postprocessing of the results, e.g., by formatting mechanisms [2]. We will discuss the indentation processing of the VIL template language as part of the content statements in Section 3.2.9.6.

Akin to programming languages, VIL templates may contain (global) variable declarations as well as sub-templates (similar to methods in object-oriented programming languages or functions in the structured programming paradigm).

Syntax:

```
//imports
//functional extensions
@advice(ivmlProjectName)
@indent(indentationSpec)
@format(formatSpec)
template name (parameterList) [extends name1] {
  //optional version specification
  //variable definitions
  //sub-template declarations
}
```

Description of syntax: The definition of a VIL template consists of the following elements:

- Optionally, imported templates or functional extensions by Java classes are listed first.
- Optional advices declaring the underlying variability models. This annotation is similar to VIL (see Section 3.1.2).
- An optional indentation annotation enabling the VIL template execution to take the actual indentation into account when processing content statements. We will detail the use of the indentation annotation in Section

3.2.9.6 along with the content statement, which actually considers indentation information.

- An optional formatting specification annotation, in particular to define the line ending of the target artifact. We will detail also the use of the format annotation in Section 3.2.9.6 along with the content statement, which actually considers indentation information.
- The keyword **`template`** defines that the following identifier *`name`* defines a new artifact instantiation template.
- The parameter list denotes the arguments a VIL template requires when being executed. Basically, a VIL-template receives the underlying variability configuration and the target artifact as parameters. If given as first parameters in this sequence, the parameters will be bound independently of their name and, thus, the parameters can be named arbitrarily. Otherwise (due to the option of additional named parameters as mentioned below), the sequence of the parameters may be arbitrary but the parameters must exactly be named "config" and "source". The arguments are subject to dynamic dispatch, i.e., either the most generic type `Artifact` may be used for the target artifact or a more specific type can be used. In the latter case, the instantiator statement in the VIL script must also pass in a type-compliant artifact instance. Additional parameters may be defined which then must be stated in the calling VIL script as named arguments.
- A VIL template may optionally extend an existing (imported) VIL template. This is expressed by **`extends`** *$name_1$*, where *`name1`* denotes the name of the extending script.
- The optional version specification, variable declarations and sub-templates are then stated within the curly brackets.

Example:

```
@advice(YMS)
template DbInit (Configuration config, Artifact target){
  /* Go on with description of the artifact instantiation
  starting with a main sub-template and possibly further
  (imported) sub-templates */
}
```

3.2.3 Version

Akin to IMVL and VIL, also the VIL template language can be tagged with an explicit version number in order to support evolution. The syntax for the version declaration is identical to VIL as discussed in Section 3.1.3.

3.2.4 Imports

The description of the instantiation of a certain artifact type may be defined in a single VIL template (possibly including sub-templates) or may be composed from reusable sub-templates specified in other (existing) scripts. Therefore, VIL templates may be imported. In order to support also the evolution of product line build specifications, also the VIL template language allows the specification of version-restricted imports. Imports make the sub-templates defined in the specified build file accessible to the importing template. The syntax of imports in VIL templates is identical to imports in the VIL scripts as discussed in Section 3.1.4. Akin to VIL scripts, cyclic imports are not allowed and shall lead to error messages.

3.2.5 Typedefs

Akin to VIL, also in the VIL template language typedefs can be defined to simplify the use of complex types. Syntax and semantics for typdefs is identical to VIL as discussed in Section 3.1.6.

3.2.6 Functional Extension

Sometimes, it is necessary to realize specific supporting functions such as calculations in terms of a programming language rather than in the template language itself. Therefore, similar to Xtend [2], the VIL template language enables external functions in terms of static Java methods, to call these methods from the template language and to use the results in VIL templates. Basically, the realizing classes are declared in VIL as extension and containing static methods are made available as they would be VIL operations[10]. However, methods with already known signatures will not be redefined.

Syntax:

```
extension name;
```

Description of Syntax: A functional extension in the VIL template language consists of the following elements:

- The keyword **extension** followed by a qualified Java *name* denoting the class to be considered. The referred class must be available to VIL through class loading. Contained static methods will be considered as extension methods for the VIL template language. Please note that the implementing method shall use only primitive Java types or (the implementation classes of the) VIL types discussed in Section 3.3.
- An extension declaration ends with a semicolon.

Example:

[10] Additional classes may require special treatment in terms of class loading, in particular in OSGi and Eclipse due to specific class loaders per bundle. In such environments, the class loader being responsible for the extension classes must be explicitly registered with the VIL runtime environment (see `de.uni_hildesheim.sse.easy_producer.instantiator.model.templateModel.ExtensionClassLoaders`).

```
extension java.lang.System;
```

The statement above makes all static Methods of the class `java.lang.System` available to VTL.

3.2.7 Types

Basically, the VIL template language is a statically typed language with some convenience in terms of postponed type checking at runtime akin to VIL. Thus, the VIL template language provides a set of formal types available for variable declarations or parameter lists. VIL template language and VIL rely on the same type system and, thus, the VIL template language provides the same types as discussed in Section 3.1.5.

3.2.8 Variables

A variable provides named access to a value of a certain type similar to variables in programming languages. The semantic of variables as well as the syntax for declaring and using them in the VIL template language is identical to VIL as discussed in Section 3.1.6 (except for the capability of defining variable values in an external file which is not available in the VIL template language).

Similar to VIL, variables may be referred in Strings such as paths or content statements. A variable reference looks like `$variableName`. Even entire VIL expressions (see Section 3.3) including variables may be given in the form `${expression}`. When applying the respective element, variable and expression references are substituted by their actual value.

3.2.9 Sub-Templates (defs)

The actual instantiation of an artifact is given in terms of sub-templates (called `def` in the concrete syntax), i.e., named functional units with parameters and return types. One specific sub-template (usually called `main`) acts as the entry point into artifact instantation. Akin to VIL, it receives the parameters of the containing template as arguments (in the same sequence).

The body of a sub-template specifies the individual steps to be executed for realizing the instantiation. Such a body may contain variable declarations, (assignment) expressions, alternatives, switch-case-statements, loops and content statements (for producing the actual content). We will first describe the syntax of templates and discuss then the statements available in sub-template bodies.

Syntax:

```
[protected] def [Type] name (parameterList) {
//variable declarations
//alternative, switch-case, loop
//content statements
}
```

Description of Syntax: A sub-template declaration consists of the following elements:

- The optional keyword **`protected`** prevents that this rule is visible from outside so that such rules cannot be used as an entry point or from other scripts.
- The keyword **`def`** indicates the definition of a sub-template.
- By default, the VIL template language aims at inferring the return type of a sub-template from the rule body. In the extreme case, individual statements may produce a rather generic value of type `Any`, which enables the use of the value without type checking at template parsing time and type checking at runtime. However, in some situations the template developer may explicitly want to do strict type checking at template parsing time. This is enabled by specifying the optional return type for a sub-template.
- The *`name`* allows identifying the sub-template for calls, template extension or as `main` entry point.
- The *`parameterList`* specifies the parameters of a sub-template in order to parameterize the instantiation operations subsumed by the respective sub-template. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Parameters must either be bound by the calling sub-template or, in case of the main entry rule, by the VIL template itself (via identical names and assignable types, the template and the main sub-template).
- The rule body is specified within the following curly brackets.

Example:

```
def main(Configuration config, Artifact target) {
    // define artifact instantiation
}

def String valueMapping (DecisionVariable var) {
    // map the value of var to a String
    // explicit type checking is enforced
}
```

The sub-template body specifies the individual steps needed to instantiate an artifact. Such a rule body may contain variable declarations, (assignment) expressions, alternatives, switch-case-statements, loops and content statements (for producing the actual content). The statements (ended by a semicolon or a statement block) given in a sub-template body are executed in the given sequence. We will discuss these individual statement types in the following subsections.

The last statement executed in a sub-template body implicitly determines the return value of a sub-template. Please note that returning a String requires an expression such as a switch (Section 3.2.9.4) or a variable declaration, as otherwise a String is recognized as a Content statement (Section 3.2.9.6).

3.2.9.1 Variable Declaration

A variable declaration within a rule body introduces a local variable shadowing rule parameters or global variables. This is in particular true for variables, which are defined within statement-blocks such as loops or alternatives. Basically, a variable declaration within a sub-template body follows the same syntax as global variable declarations discussed in Section 3.2.8.

3.2.9.2 Expression Statement

Expressions such as value calculations or execution of artifact operations may be used within a sub-template body as a guard expression or as a variable assignment. Please note that we will detail the VIL expression language in Section 3.3, as the expression language is common to both, the VIL instantiation language and the VIL template language. Thus, guard expressions and variable assignments as discussed in Section 3.1.9.2 are similarly available in the VIL template language. Further, similar call types as well as their (resolution) semantic as discussed in Section 3.1.9.3 are available in the VIL template language (of course, rule calls are replaced by template calls and operating system calls by calls to functional extensions). Template calls may be recursive. In the VIL template language, the resolution sequence is

1) Template calls
2) Artifact, configuration type and basic type operations
3) Functional extension calls

3.2.9.3 Alternative

In the VIL template language, alternatives allow choosing among different ways of instantiating an artifact, i.e., upon evaluating a condition the appropriate alternative to execute is determined.

The (return) type of an alternative is either the common type of all alternatives or `Any`. The return value of an alternative is determined by the last statement executed in the alternative selected by the (condition) expression.

Syntax:

```
if (expression) ifStatement
if (expression) ifStatement else elseStatement
```

Description of Syntax: An alternative statement consists of the following elements:

- The keyword **`if`** indicates the beginning of an alternative statement.
- The *`expression`* given in parenthesis determines whether the if-part (condition is evaluated to true) or the else-part (condition is evaluated to false) is executed. If *`expression`* cannot be evaluated, the alternatives will be evaluated.

- The *ifStatement* (or statement block enclosed in curly braces) is being executed when the *expression* is evaluated to true. A single statement must be terminated by a semicolon.
- The **else** part is optional, i.e., if else is used in an alternative, a following *elseStatement* is required. As usual, a dangling else is bound to the innermost alternative.
- The *elseStatement* (or statement block enclosed in curly braces) is executed if the *expression* is evaluated to false. Again, a single statement must be terminated by a semicolon.

Example:

```
if (config.variables().size() > 0) {
    // work on config
} else {
    // produce an empty artifact
}
```

3.2.9.4 Switch

The switch statement in the VIL template language is for (dynamically) mapping configuration elements to artifact elements rather than for influencing the control flow (as it is the case for the alternative statement). However, in case of larger mappings with (more or less) static content, we suggest using a map variable (see Section 3.1.5.4).

The (return) type of an alternative is either the common type of all cases or `Any`. The return value of an alternative is determined by the last statement executed in the alternative selected by the (condition) expression.

Syntax:

```
switch (expression) {
  expression1 : expression2,
  expression3 : expression4,
  [default : expression5]
}
```

Description of Syntax: An alternative statement consists of the following elements:

- The keyword **switch** indicates the beginning of a switch statement. It is followed by an *expression* to switch over and the individual cases in a block of curly brackets.
- A case consists of an expression (*expression*$_1$ or *expression*$_3$ above) to be matched against expression. If an individual match succeeds, the related value expression will be evaluated and determines the (result) value of the switch statement. The implicit variable `VALUE` may be used

within the value expression in order to refer to the evaluated value of *expression.*

- Optionally, a **`default`** case can be given which is considered if none of the previous cases matches. In that case, the expression stated behind the default will be evaluated and determines the (result) value of the switch statement.

Example:

```
switch (var.name()) {
    "forkNumber" : VALUE + var.intValue() - 1,
    "cpuNumber" : var.stringValue()
    //go on with further cases and a default value
    //if required
}
```

3.2.9.5 Loop

The for-statement in VIL enables the defined repetition of statements. Basically, it is rather similar to an iterator-loop in Java.

Syntax:

for (Type *var* : *expression*) *statement*

for (Type *var* : *expression*, *expression*$_1$) *statement*

for (Type *var* : *expression*, *expression*$_1$, *expression*$_2$) *statement*

Description of Syntax: A loop statement consists of the following elements:

- The keyword **`for`** indicates the beginning of a for-loop statement. It is followed by a parenthesis defining the loop iterator, i.e., a variable to which successively all values of the *expression* are assigned. Therefore, *expression* must either evaluate to a set or a sequence.
- The statement (or statement block enclosed in curly braces) is then executed for each element in the collection specified by *expression* while the iterator *var* is successively assigned to each individual value in the collection.
- An optional separator expression *expression*$_1$ which is emitted (without line end) at the end of each iteration if further iterations will happen. Such a separator expression simplifies generating value lists or similar target artifact concepts.
- A second optional separator expression *expression*$_2$ which may only be stated if *expression*$_1$ is given. *expression*$_2$ is emitted (without line end) at the end of the last iteration. Such a separator expression simplifies generating value lists or similar target artifact concepts.

Example:

```
for (DecisionVariable var: config.variables()) {
    //operate on var
}
```

Akin to VIL, VTL supports the implicit conversion of a decision variable to its contained variables, i.e., we may omit the `variables()` operation.

```
for (DecisionVariable var: config, ", ") {
    '${var.name()}'
    // (the separator will be added if needed)
}

for (DecisionVariable var: config, ",", ";") {
    '${var.name()}'
    //(the separator will be added if needed,
    //the second separator will be printed after the final
    //iteration)
}
```

3.2.9.6 Content

The content statement is used to generate the content of the target artifact. Basically, all characters given in a String (enclosed in a pair of apostrophs or quotes including appropriate Java escapes and line breaks) are emitted as output to the result artifact. Content statements executed in the course of template evaluation according to the control flow make up the entire content of the target artifact (fragment). A content statement may consist of multiple lines as part of the content. Thereby, variable references or IVML expressions are substituted as described for variables in Section 3.2.8.

Without further consideration, also the indentation whitespaces for pretty-printing a VIL-template will be taken over into the resulting artifact. In order to provide more control about the formatting, the annotation **`@indent`** allows specifying the number of whitespaces used as one indentation step (value `indentation`), the number of whitespaces to be considered in tabulator emulation (value `tabEmulation`) and also how many additional whitespaces (value `additional`, default is `1`) are used to indent the content statement, i.e., whether the following lines after the lead in character are further subject to indentation or not. Further whitespaces in the content are considered as formatting of the content itself and are taken over into the artifact. In addition, an optional numerical value can be specified at each content statement in order to programmatically indent the configuration by the given number of whitespaces.

In some cases, it is required to explicitly define the line ending of the target artifact, e.g., when operating system scripts shall be generated or manipulated. Therefore, the annotation **`@format`** allows specifying the formatting. Akin to **`@indent`**, the formatting is defined as key value pairs. Currently, VTL supports exactly the key **`lineEnd`**, which receives a String value defining the line end, e.g., **`@format(lineEnd = "\r\n")`** for a forced Linux line end.

Syntax:

```
"text"

print "text"

"text" | expression;

'text' | expression;
```

Description of Syntax: A content statement consists of the following elements:

- The optional keyword **`print`**, which indicates that no line end shall be emitted at the end of the content. If absent, implicitly a print-line is performed. The print form of the content statement is helpful in combination with the separator expression in loops.
- The lead in / lead out marker (apostrophe or quote) indicates the content statement and marks the actual content. Two forms are used to enable the use of the opposite character in content. A content statement may cover multiple lines of content.
- An optional indentation expression can be indicated by the pipe symbol and followed by a numerical *`expression`*. The numerical *`expression`* determines the amount of whitespaces to be used as mandatory indentation prefix for each individual line of the actual content statement. Indentation is only considered if the expression evaluates to a positive number, i.e., 0 or a negative number may be given, but this is then ignored. Only if the indentation expression (may be a constant or a true expression) is specified, a semicolon is required.

Example:

```
'CREATE DATABASE ${var.name()}'
'CREATE TABLE data' | 4;
```

While the first content statement emits the text with expression substitution as it is, the second content statement indents the text to be emitted by 4 whitespaces.

3.3 VIL Expression Language

In this section, we will define the syntax and the semantics of the VIL expression language, which is common to the VIL instantiation language and the VIL template language. Similar to IVML, expressions in the VIL languages are inspired by OCL. Thus, most of the content in this section is taken from OCL [6] or the IVML language specification [3, 7] and adjusted to the need, the notational conventions, and the semantics of the VIL languages, in particular also regarding the syntax of the iterators and the absence of quantors in VIL.

3.3.1 Reserved Keywords

Keywords in the VIL expression language are reserved words. That means that the keywords must not occur anywhere in an expression as the name of a rule, a template or a variable. The list of keywords is shown below:

- **`and`**
- **`callOf`**
- **`false`**
- **`new`**
- **`not`**
- **`or`**
- **`sequenceOf`**
- **`setOf`**
- **`super`**
- **`true`**
- **`xor`**
- **`null`**

In order to increase reuse among the VIL languages, the VIL expression language also provides the definition of common language concepts such as variable declarations and parameter lists. The related keywords were already listed in Section 3.1.1 and 3.2.1, respectively.

3.3.2 Prefix operators

The VIL expression language defines two prefix operators, the unary

- Boolean negation '**`not`**' and its alias '!'.
- Numerical negation '–' which changes the sign of a Real or an Integer.

Operators may be applied to constants, (qualified) variables or return values of calls.

3.3.3 Infix operators

Similar to OCL, in VIL the use of infix operators is allowed. The operators '+,' '–,' '`*`.' '/,' '<,' '>,' '<>' '<=' '>=' are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

```
a + b
```

is conceptually equal to the expression:

```
a.+(b)
```

that is, invoking the "+" operation on *a* (the *operand*) through the dot-access notation with *b* as the parameter to the operation. The infix operators defined for a type must have exactly one parameter. For the infix operators '<,' '>,' '<=,' '>=,' '<>,' '**and**,' '**or**,' '**xor**', the return type is Boolean.

Operators may be applied to constants, (qualified) variables or return values of calls.

Please note that, while using infix operators, in VIL Integer is a subclass of Real. Thus, for each parameter of type Real, you can use Integer as the actual parameter. However, the return type will always be Real. We will detail the operations on basic types in Section 3.4.2.

3.3.4 Precedence rules

The precedence order for the operations, starting with highest precedence, in IVML is:

- dot operations: '`.`' (for operation calls in object-oriented style)
- unary '**not**', !(alias for **not**) and unary minus '–'
- '*' and '/'
- '+' and binary '–'
- '<', '>', '<=', '>='
- '==' (equality), '<>', '!=' (alias for '<>')
- '**and**', '**or**' and '**xor**'

'`(`' and '`)`' can be used to change precedence.

3.3.5 Datatypes

All artifacts defined by the extensible VIL artefact model as well as the various built-in types are available to the expression language and may be used in expressions. IVML elements are mapped into VIL via IVML qualified names. Figure 1 illustrates the VIL type hierarchy (not detailing the IVML integration through the configuration types). Below, we discuss the use of datatypes.

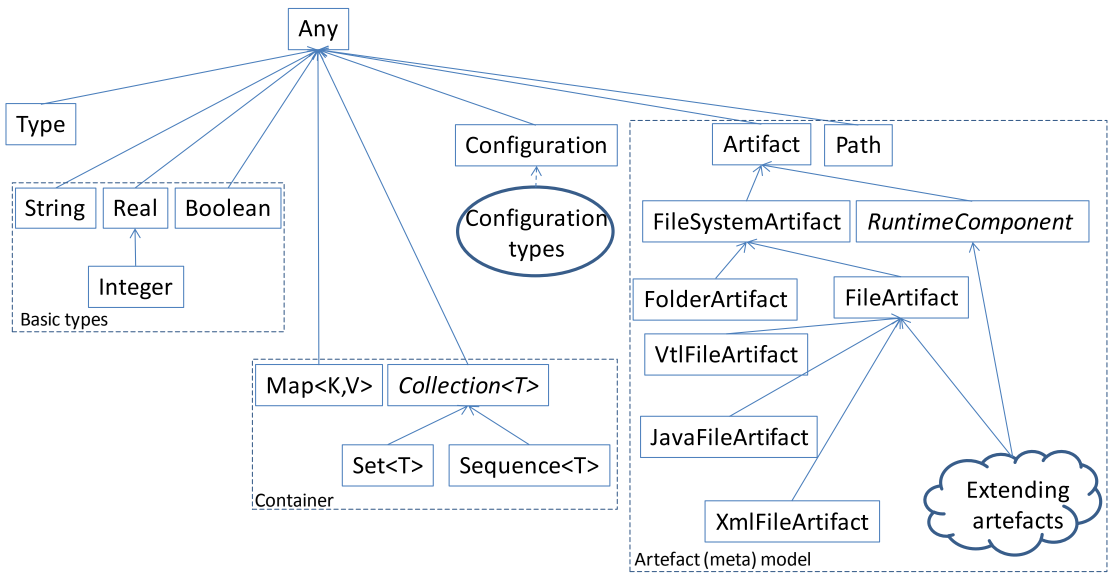

Figure 1: Overview of the VIL type system

3.3.6 Type conformance

Type conformance in IVML constraints is inspired by OCL (cf. OCL section 7.4.5):

- `Any` is the common superclass of all types. All types comply with `Any`. However, `Any` is not directly available to the user and used internally to denote expressions of unknown type to be resolved and checked while execution time of the specifying script.
- Each type conforms to its (transitive) supertypes. Figure 1 depicts the IVML type hierarchy.
- Type conformance is transitive.
- The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real (actually the type Integer is considered as a subclass of Real).
- Containers are parameterized types regarding the contained element type. Containers comply only if they are of the same container type and the type of the contained elements complies (checked before script evaluation time) or if no type parameters are given (deferred to script evaluation time).

3.3.7 Side effects

In contrast to OCL, some constraint expressions in IVML may lead to side effects, in particular to modifications of artifacts and artifact fragments.

3.3.8 Undefined values

Basically, variables and IVML qualified names, i.e., links into a variability model, may be undefined. During evaluation of expressions at script or template execution time, undefined (sub-)expressions are ignored and do not lead to failing rules or sub-templates. This is in particular true for VTL content statements (see Section 3.2.9.6), which are not emitted if any subexpression is undefined.

3.3.9 Null

An IVML variable may have the explicit value `null`, i.e., it is configured with `null`. In VIL / VTL, the keyword `null` represents the `null` value in IVML. This can be helpful to figure out, whether a variable was configured with null as shown below (for VTL):

```
if (var == null) {
    // ...
}
```

From the perspective of the type system[11], `null` is a specific value of `Any`, i.e., it is not a pointer to nowhere as in many programming languages. Thus, `null` is defined and does not lead to undefined values and ignored expressions (see Section 3.3.8). Please note that a decision variable configured with `null` is not undefined from the point of view of IVML as it has a value.

3.3.10 Collection operations

The VIL artifact model and the IVML integration into VIL in terms of the `Configuration` type define many operations returning collections. Operating with collections is specifically meant to enable a flexible and powerful way of accessing information and for deriving artifacts.

Practically, collections in VIL are sets, sequences, or maps as introduced in Section 3.1.5.4. Collections are a basis for iterations in the VIL template language (Section 3.2.9.5) as well as for joins and map iterations in VIL (Section 3.1.9.6). Therefore, we defined a set of basic operations on the artifact types and the `Configuration` providing convenient access through selection and filtering. This is expressed in terms of iterator expressions, which are called through the "->" notation rather than the dot-notation ".". One example is the **select** operation, which returns all elements of a collection complying to a certain Boolean selection expression. An example for an expression with an iterator variable[12] is

```
configuration.variables()->select(i|i.name().length()== 10)
```

or with explicit types (of course the type must match the collection type)

```
configuration.variables()->select(
  DecisionVariable i|i.name().length()== 10)
```

Both expressions return those decision variables with a name of length 10. In addition to select, also a generic sort operation is provided for sequences.

Other examples for collection iterator expressions are the **collect** and **apply** operations also known from IVML / OCL. While collection returns the value of an expression for each element of a collection, apply allows defining arbitrary iterative expressions. Following the example shown above, the *basic form* of apply is

[11] Please note that the notation (**null** == var) is (currently) not considered as a valid operation as `Any` does not define operations.

[12] The old notation using the generic iterator name `ITER` still works for compatibility. The new notation is akin to IVML, but the explicit type of an iterator variable is stated in VIL/VTL before the name of the variable. The syntax allows for multiple iterators, but currently only one iterator variable is supported.

```
configuration.variables()->apply(Integer r = 0; i|
  r = r + i.name().length())
```

The first declarator defines the aggregator, the second the iterator over the collection, while the expression modifies the aggregator successively. In this form, the result type of apply corresponds to the type of the aggregator and its value is the value of the aggregator at the end of the iteration. Please note that aggregators must be initialized with a default value, while iterators must not have a default value (implicitly taken from the collection an iterator is applied to). The short form of apply does not define an aggregator and, thus, just iterates over the collection without result to be returned. Assuming that `files` is a collection of `FileArtifact`, then the following expression deletes all those artifacts.

```
files->apply(f|f.delete())
```

Some operations can provide implicit collections so that iterator variables and dependent expressions can be applied similarly. One example is the deleteStatement operation provided by the JavaMethod (Section 3.5.2). Given such a Java method, one can delete all JavaCalls matching the condition given by the iterator expression as shown below:

```
method->deleteStatement(JavaCall c |
  c.type() == "EASyLogger" and c.name() == "exception");
```

3.3.11 Dynamic extension of the type system through IVML

Basically, VIL / VTL are loosely integrated with the variability model, i.e., generic VIL / VTL types allow accessing IVML variables and configurations, e.g., via the name of a variable. The generic access is beneficial, if the variability model itself may change dynamically (a potential future capability) or to iterate over the structure of the variability model and derive artifact based on the structure. However, in standard situations having a fixed variability model, using the generic VIL / VTL types may be inconvenient.

Until version 0.9 of VIL / VTL, the `@advice` annotation enabled to use IVML identifiers directly in VIL / VTL instead of strings for variable names or enumeration values. This supports the specification of VIL / VTL scripts as external names are checked by the editor. Since version 0.92, the `@advice` annotation makes also IVML types available to VIL / VTL. For example, let

```
project Demo {
  compound MyCompound {
    Integer slotA;
  }
  MyCompound myCompound;
}
```

be an IVML variability model. Accessing the variable `slotA` in `myCompound` looks in VIL/VTL using the generic types as follows

```
DecisionVariable comp = config.byName(“myCompound”);
Integer i = v.byName(“slotA”);
```

or after using `@advice(Demo)` also

```
DecisionVariable comp = config.byName(myCompound);
Integer i = v.byName(MyCompound::slotA);
```

Using `@advice(Demo)` in VIL/VTL version version 0.95 [13]

```
Demo d = config;
Integer i = d.myCompound.slotA;
```

Similarly, also the project annotations can be accessed. Moreover, and this was the initial motivation for extending the `@advice` annotation, IVML types can directly be used in rule and template signatures in order to exploit dynamic dispatch (see Section 3.1.9.3) to process refined IVML compounds in an extensible manner. As a result, differentiating types can easily be expressed in terms of rules or templates using the same signature but specialized parameter types for all involved IVML types.

In more detail, the following rules apply if an `@advice` annotation is used:

- IVML types are mapped transparently to VIL / VTL types. IVML types from projects specified in `@advice` annotations are mapped with their simple type name (if the name has not been defined before in VIL) and with their qualified name. In particular, contained variables are represented as fields of the specific IVML type (in contrast to the generic integration via `DecisionVariable`). Please note that the value of fields cannot be changed in VIL.
- IVML projects are mapped transparently to VIL / VTL types to provide seamless access to the top-level IVML variables. A VIL / VTL configuration instance can directly be assigned to a VIL / VTL variable of a project type. Thereby, the configuration is projected to the respective (sub-)project.
- Annotations of an IVML project are mapped as fields of the specific types defined by the IVML project. Annotations defined on individual variables are mapped to the type. However, inconsistent individual definitions will be mapped only once (for the first type in sequence). In case that individual annotations with different types are used, the generic VIL / VTL types shall be used to access these annotations. Please note that the value of annotations cannot be changed in VIL.
- All types which are dynamically introduced to VIL / VTL via `@advice` annotations define the (generic) operations from `DecisionVariable` (see Section 3.4.5.3 for details).

Due to implicit conversions, one can deliberately switch between the specific IVML types and the generic types. However, it is advisable to follow one style in order to simplify the VIL/VTL specifications. Akin to the lightweight checking of identifiers introduced by `@advice`, i.e., unknown identifiers are reported by the editors as a warning, also unknown types and operations on unknown types are not checked strictly and indicated by a warning. However, unknown types are considered as undefined expressions (see Section 3.3.8) and, thus, not evaluated.

[13] To improve and clarify the mapping, the old mapping implemented in versions 0.92 to 0.94 has been disabled.

3.4 Built-in operations

Similar to OCL and IVML, all operations in VIL are defined on individual types and can be accessed using the "." operator, such as `set.size()`. However, this is also true for the equality, relational, and mathematical operators but they are typically given in alternative infix notation, i.e., 1 + 1 instead of 1.+(1) as stated in Section 3.3.3. Further, the unary negation is typically stated as prefix operator. Due to the VIL artifact model, the integration with IVML and the specific purpose of variability instantiation, the VIL types define a different set of operations than OCL/IVML[14].

In this section, we denote the actual type on which an individual operation is defined as the *operand* of the operation (called *self* in OCL). The parameters of an operation are given in parenthesis. Further, we use in this section the Type-first notation to describe the signatures of the operation.

Please note that those operations starting with "get" (Java-getters) are also available with their short name without "get" in order to simplify script and template creation, e.g., the `getName()` operation is also available as its aliased operation `name()`. We will make this explicit by listing both operation signatures.

3.4.1 Internal Types

3.4.1.1 Any

`Any` is the most common type in the VIL type system. All types in VIL are type compliant to `Any`. In particular, `Any` is used as type if the actual type is unknown at parsing time and shall be determined dynamically at runtime. Therefore, `Any` can be assigned to variables of any type (but no specific operations can be executed on `Any`). `Any` can be converted automatically into a `String`.

3.4.1.2 Type

Type represents type expressions themselves and enables the type-generic selection type-compliant elements from collections.

3.4.2 Basic Types

In this section, we detail the operations for the basic VIL types.

3.4.2.1 Real

The basic type `Real` represents the mathematical concept of real numbers following the Java range restrictions for double values. Note that `Integer` can automatically be converted to `Real`.

- **Real + (Real r)**
  The value of the addition of *self* and the *operand*.
- **Real - (Real r)**
  The value of the subtraction of *r* from the *operand*.
- **Real * (Real r)**

[14] An extended set of operations will be defined by future extensions of VIL.

The value of the multiplication of the *operand* and *r*.

- **Real - ()**
  The negative value of the *operand*.
- **Real / (Real r)**
  The value of the *operand* divided by *r*. Leads to an evaluation error if *r* is equal to zero.
- **Boolean < (Real r)**
  True if the *operand* is less than *r*.
- **Boolean > (Real r)**
  True if the *operand* is greater than *r*.
- **Boolean <= (Real r)**
  True if the *operand* is less than or equal to *r*.
- **Boolean >= (Real r)**
  True if the *operand* is the same as *r*.
- **Boolean != (Real r)**
  True if the *operand* is not the same as *r*.
- **Boolean <> (Real r)**
  True if the *operand* is not the same as *r*.
- **Boolean == (Real r)**
  True if the *operand* is the same as *r*.
- **Real abs()**
  The absolute value of the *operand*.
- **Integer floor ()**
  The largest integer that is less than or equal to the *operand*.
- **Integer ceil()**
  The closest integer value that is greater or equal to the *operand*.
- **Integer round()**
  The integer that is closest to *the operand*. When there are two such integers, the largest one.
- **Boolean isNaN()**
  Returns whether the *operand* is not a number.
- **Boolean isFinite()**
  Returns whether the *operand* is a finite real.
- **Boolean isInfinite()**
  Returns whether the *operand* is an infinite real.
- **Real random()**
  Returns a random number 0 and 1.0 (exclusive). No operand is needed.

3.4.2.2 Integer

The standard type `Integer` represents the mathematical concept of integer numbers following the Java range restrictions for integer values. Note that `Integer` is a subclass of `Real`.

- **Integer + (Integer i)**
  The value of the addition of the *operand* and *i*.
- **Integer - (Integer i)**

The value of the subtraction of *i* from the *operand*.

- **Integer * (Integer i)**
  The value of the multiplication of the *operand* and *i*.
- **Real / (Integer i)**
  The value of the *operand* divided by *i*. Leads to an evaluation error if *i* is equal to zero.
- **Boolean < (Integer i)**
  True if the *operand* is less than *i*.
- **Boolean > (Integer i)**
  True if the *operand* is greater than *i*.
- **Boolean <= (Integer i)**
  True if the *operand* is less than or equal to *i*.
- **Boolean != (Integer i)**
  True if the *operand* is not the same as *i*.
- **Boolean <> (Integer i)**
  True if the *operand* is not the same as *i*.
- **Boolean >= (Integer i)**
  True if the *operand* is greater than or equal to *i*.
- **Boolean == (Integer i)**
  True if the *operand* is the same as *i*.
- **Integer - ()**
  The negative value of the *operand*.
- **Integer abs()**
  The absolute value of the *operand*.
- **Integer randomInteger()**
  Returns a random integer between 0 and 1.0 (exclusive). No operand is needed.
- **Integer randomInteger(Integer m)**
  Returns a random integer between 0 and m-1 (exclusive) using m as operand (to distinguish from `randomInteger()`.

Conversions: `Integer` values can automatically be converted to `Real` values.

Boolean

The basic type `Boolean` represents the common true/false values.

- **Boolean == (Boolean a)**
  True if the *operand* is the same as *a*.
- **Boolean <> (Boolean a)**
  True if the *operand* is not the same as *a*.
- **Boolean != (Boolean a)**
  True if the *operand* is not the same as *a*.
- **Boolean or (Boolean b)**
  True if either *self* or *b* is true.
- **Boolean xor (Boolean b)**
  True if either *self* or *b* is true, but not both.
- **Boolean and (Boolean b)**

True if both *b1* and *b* are true.

- **Boolean not ()**
  True if *self* is false and vice versa.
- **Boolean ! ()**
  True if *self* is false and vice versa.

3.4.2.3 String

The standard type `String` represents strings, which can be ASCII.

- **Integer length ()**
  The number of characters in the *operand*.
- **String + (Any s)**
  The concatenation of the *operand* and (the Java String representation of) *s*. In particular, this operation concatenates two Strings.
- **String + (Path p)**
  The concatenation of the *operand* and the string representation of path *p*.
- **String substring(Integer s, Integer e)**
  Returns the substring of the *operand* from *s* (inclusive) to *e* (exclusive). *operand* is returned in case of any problem, e.g., positions exceeding the valid index range.
- **String replace(String s, String r)**
  Returns *operand* with all substrings *s* replaced by *r*.
- **String substitute(String s, String r, String p)**
  Returns *operand* with all substrings matching the Java regular expression *r* substituted by *p*.
- **Boolean <> (String s)**
  True if the *operand* is not the same as *s*.
- **Boolean != (String s)**
  True if the *operand* is not the same as *s*.
- **Boolean == (String s)**
  True if the *operand* is the same as *s*.
- **Boolean matches (String r)**
  Returns whether the *operand* matches the regular expression *r*. Regular expressions are given in the Java regular expression notation. For example, the following operation will check whether `mail` is a valid e-mail-address:
  ```
      mail.matches([\w]*@[\w]*.[\w]*);
  ```
- **sequenceOf(String) split (String r)**
  Splits the *operand* along the regular expression r. Regular expressions are given in the Java regular expression notation. If r does not match an empty collection is returned.
- **String toUpperCase()**
  Turns the operand into a `String` consisting of upper case characters.
- **String toLowerCase()**
  Turns the *operand* into a `String` consisting of lower case characters.
- **String firstToUpperCase()**
  Turns the first character of *operand* into an upper case character.
- **String firstToLowerCase()**

Turns the first character of *operand* into a lower case character.

- **String toIdentifier()**
  Turns the operand into a typical (Java) programming language identifier, e.g., by removing illegal characters such as spaces. Cases of the individual characters will not be changed. The result may be empty.
- **Integer toInteger ()**
  Converts the *operand* to an Integer value.
- **Real toReal ()**
  Converts the *operand* to a Real value.

3.4.3 Container Types

This section defines the operations of the collection types. VIL defines one abstract collection type `Collection` and two specific collections, namely `Set` and `Sequence`. All collection types are parameterized by one parameter. Below, 'T' will denote the parameter for the collection types. A concrete collection type is created by substituting a type for the parameter T. So a collection of integers is referred in VIL by `setOf(Integer)`. Although the keyword `collectionOf` does not exist, we will use it in this section to denote types of Collection (Section 3.4.3.1)

In addition, VIL defines the type `Map`, an associative container, which allows relating keys to values.

3.4.3.1 Collection

`Collection` is the abstract super type of all collections in VIL.

- **Integer size ()**
  The number of elements in the collection *operand.*
- **Boolean includes (T object)**
  True if *object* is an element of *operand*, false otherwise.
- **Boolean excludes (T object)**
  True if *object* is not an element of *operand*, false otherwise.
- **Integer count (T object)**
  The number of times that *object* occurs in the collection *operand*.
- **Boolean ==(collectionOf(Type) c)**
  Returns whether *operand* contains the same elements than *c*, for ordered collections such as `Sequence` also whether the elements are given in the same sequence.
- **Boolean equals(collectionOf(Type) c)**
  Returns whether *operand* contains the same elements than *c*, for ordered collections such as `Sequence` also whether the elements are given in the same sequence.
- **Boolean remove (T e)**
  Removes e from *operand* and returns whether *e* was removed.
- **T add (T e)**
  Adds the element e to the set *operand*.
- **Boolean isEmpty ()**
  Is the *operand* the empty collection?

- **Boolean isNotEmpty ()**
  Is the *operand* not the empty collection?

3.4.3.2 Set

The type `Set` represents the mathematical concept of a set. It contains elements without duplicates. Set inherits the operations from `Collection`. For convenience, we added some modifying operations, which may not be supported in certain settings (unmodifiable set), in particular in the context of operation types.

- **sequenceOf(T) asSequence() / toSequence ()**[15]
  Turns *operand* into a sequence.
- **T projectSingle()**
  In case of an *operand* with one element, return that element. Otherwise, nothing will be returned and subsequent expressions may fail.
- **setOf(Type) selectByType (Type t)**
  Returns all those elements of *operand* that are type compliant to *t*.
- **setOf(T) union (setOf(T) s)**
  The union of *operand* and *s*.
- **setOf(T) intersection (setOf(T) s)**
  The intersection of *operand* and *s* (i.e., the set of all elements that are in both *operand* and *s*).
- **setOf(T) excluding (collectionOf(T) s)**
  Returns a subset of *operand*, which does not include the elements in *s*.
- **setOf(T) including (collectionOf(T) s)**
  Returns the union of *operand* and *s* (without duplicates).
- **remove (T e)**
  Removes e from *operand* and returns whether *e* was removed. Does not affect *operand* if *operand* is not modifiable.
- **T add (T e)**
  Adds the element e to the set *operand*. Does not affect *operand* if *operand* is not modifiable.
- **apply (Expression e) / A apply(Expression e)**
  Applies *e* to all elements in *operand*. Without aggregating declarator, nothing is returned and just a loop over all elements is executed. If an aggregator is declared, the operation returns a result of the type of the aggregator (denoted as *A* above).
- **setOf(T) select (Expression e)**
  Returns the elements in *operand*, which comply with the iterator *e*.
- **setOf(A) collect (Expression e)**
  Returns the results of applying the iterator expression *e* to all elements in *operand*.

[15] “toSequence” was the initial operation, “asSequence” was added for compatibility with IVML/OCL. “toSequence” may be removed in one of the next versions.

3.4.3.3 Sequence

A `Sequence` is a `Collection` in which the elements are ordered. An element may be part of a `Sequence` more than once. `Sequence` inherits the operations from `Collection`. For convenience, we added some modifying operations, which may not be supported in certain settings (unmodifiable sequence), in particular in the context of operation types.

- **T get (Integer index)**
  Returns the element of *operand* at position *index*. *index* must be valid, i.e., not negative and less than size().
- **T [Integer index]**
  The []-operator returns the element of *operand* at position *index*. *index* must be valid, i.e., not negative and less than size().
- **T first()**
  Returns the first element of *operand*.
- **T last()**
  Returns the last element of *operand*.
- **Boolean ==(collectionOf(Type) c)**
  Returns whether *operand* contains the same elements in the same sequence than *c*.
- **Boolean equals(collectionOf(Type) c)**
  Returns whether *operand* contains the same elements in the same sequence than *c*.
- **Integer indexOf(T e)**
  Returns the index of e in *operand*. If e does not exist in *operand*, -1 will be returned.
- **setOf(T) asSet() / toSet ()**[16]
  Turns *operand* into a set (excluding duplicates).
- **sequenceOf(T) selectByType (Type t)**
  Returns all those elements of *operand* that are type compliant to *t*.
- **sequenceOf(T) excluding (collectionOf(T) s)**
  Returns a subset of *operand*, which does not include the elements in *s*.
- **sequenceOf(T) append(collectionOf(T) s)**
  Returns the result of appending *s* to *operand* (possibly including duplicates). Does not affect operand rather than it returns a new sequence.
- **remove (T e)**
  Removes the first occurrence of *e* from *operand* and returns whether *e* was removed. Does not affect *operand* if *operand* is not modifiable.
- **T add (T e)**
  Adds the element e to the set *operand*. Does not affect *operand* if *operand* is not modifiable.

[16] “toSet” was the initial operation, “asSet” was added for compatibility with IVML/OCL. “toSet” may be removed in one of the next versions.

- **sequenceOf(T) select (Expression e)**
  Returns the elements in *operand* which comply with the iterator expression *e*.
- **sequenceOf(A) collect (Expression e)**
  Returns the results of applying the iterator expression *e* to all elements in *operand* in the sequence of *operand*.
- **iterate (Expression e)**
  Applies *e* to all elements in *operand*.
- **sequenceOf(T) revert()**
  Reverts the sequence in *operand* and returns a copy.
- **sequenceOf(T) sort(Expression e)**
  Sorts the elements in *operand* according to the respective values defined by the iterator expression *e*. In case of numerical values, numerical order is considered, else lexicographical order.
- **mapOf(T, U) mapSequence(sequenceOf(U) o)**
  Maps the elements of *operand* against those in o in the given sequence and returns the mapping of equal elements between both.
- **mapOf(T, U) mapAny(sequenceOf(U) o)**
  Maps the elements of *operand* against those in o regardless of their sequence and returns the mapping of equal elements between both.
- **sequenceOf(Integer) createIntegerSequence(Integer s, Integer e)**
  Creates a sequence of integers from s to e. If e is smaller than s, an empty sequence is created[17].

3.4.3.4 Map

The `Map` type represents an associative container, which is parameterized over the type of keys `K` and the type of values `V`. For convenience, we added some modifying operations, which may not be supported in certain settings (not modifiable map), in particular in the context of operation types.

- **V get (K k)**
  Returns the mapping for key *k* in *operand* at position *index*. In case that the mapping does not exist, the expression remains undefined. Use **containsKey** to avoid this.
- **V [K k]**
  The []-operator returns the value assigned to the given key *k* in *operand*. In case that the mapping does not exist, the expression remains undefined. Use **containsKey** to avoid this.
- **add(K k, V v)**
  Adds the given key-value mapping (*k*,*v*) to the map and overrides existing value for key *k*. Does not affect *operand* if *operand* is not modifiable.
- **Boolean containsKey(K k)**
  Returns whether the *operand* contains *k* as key for a mapping.

[17] Please note that this is currently realized by a real enumeration of the values, i.e., extensive sequences shall may not be created. Arbitrary sequences will be targeted in future.

- **setOf(K) getKeys() / keys()**
  Returns the keys of *operand* as a not modifiable set.
- **Integer size()**
  Returns the size of the *operand* in terms of the number of entries.
- **remove(K k)**
  Removes the mapping for key *k*. Does not affect *operand* if *operand* is not modifiable.

Sequences which contain sequences with exactly two entry types matching the types of `Map` can be converted automatically into a `Map` instance.

3.4.3.5 Iterator

`Iterator` is an internal parameterized type that can be used to allow extending types to indirectly return a collection to iterate over, e.g., via map (Section 3.1.9.6) or for (Section 3.2.9.5). This is useful, if the implementing type shall not or cannot use the VIL types directly, e.g., language extensions for rt-VIL (Section 3.6).

3.4.4 Version Type

The version type is an internal type (actually not supported as a regular type for variables) for defining version constraints. Using the type name "Version" is discouraged. Thus, the version type supports only the following operations:

- **Boolean == (Version v)**
  Evaluates to true if *operand* and *v* denote the same version.
- **Boolean < (Version v)**
  True if the *operand* is less than *v*.
- **Boolean > (Version v)**
  True if the *operand* is greater than *v*.
- **Boolean <= (Version v)**
  True if the *operand* is less than or equal to *v*.
- **Boolean >= (Version v)**
  True if the *operand* is greater than or equal to *v*.
- **Boolean == (AnyType a)**
  True if the *operand* is the same as *a*. This operation is interpreted as a value assertion if it is used standalone (empty implication) or on the right side of an implication. It is interpreted as an equality test if used on the left side of an implication.
- **Boolean <> (AnyType a)**
  True if the *operand* is different from *a*.
- **Boolean != (AnyType a)**
  True if the *operand* is a different object from *a*. Alias for <>.

3.4.5 Configuration Types

Configuration types realize the integration with IVML. As the VIL languages are intended for variability instantiation rather than variability modelling, the Configuration types neither support changing the underlying IVML model nor its configuration.

3.4.5.1 IvmlElement

The `IvmlElement` is the most common configuration type. All configuration types discussed in this section are subclasses of `IvmlElement`. Further, this type represents the IVML identifiers used in VIL scripts or templates.

- **Boolean ==(IvmlElement i)**
  Returns whether *operand* is the same IvmlElement as *i*.
- **String getName () / String name ()**
  Returns the (unqualified) name of the *operand*.
- **String getQualifiedName () / String qualifiedName ()**
  Returns the unqualified name of the *operand*.
- **String getType () / String type ()**
  Returns the (unqualified) name of the type of the *operand*.
- **String getQualifiedType () / String qualifiedType ()**
  Returns the unqualified name of the type of the IVML element.
- **IvmlElement getElement(String n) / IvmlElement element (String n)**
  Returns the (nested) element of the *operand* with given name *n*.
- **Annotation getAttribute (String n) / Annotation attribute (String n)**[18]
  Returns the annotation of the *operand* with given name *n*.
- **Annotation getAnnotation (String n) / Annotation annotation (String n)**
  Returns the annotation of the *operand* with given name *n*.
- **Boolean isNull()**
  Returns whether the value of the *operand* is null (in the sense of IVML, see Section 3.3.9).
- **Any getValue () / Any value ()**
  Returns the (untyped) configuration value of the *operand*.
- **String getStringValue () / String stringValue ()**
  Returns the configuration value of the *operand* as a `String` in case that the underlying IVML decision variable is of type `String` or the `String` representation of the value in any other case.
- **Boolean getBooleanValue () / Boolean booleanValue ()**
  Returns the configuration value of the *operand* as a `Boolean`, but is undefined if the underlying IVML decision variable is not of type `Boolean`.
- **Integer getIntegerValue () / Integer integerValue ()**
  Returns the configuration value of the *operand* as an `Integer`, but is undefined if the underlying IVML decision variable is not of type `Integer`.
- **Real getRealValue () / Real realValue ()**
  Returns the configuration value of the *operand* as a `Real`, but is undefined if the underlying IVML decision variable is not of type `Real`.
- **EnumValue getEnumValue () / Enum enumValue ()**
  Returns the configuration value of the *operand* as an `EnumValue`, but is undefined if the underlying IVML decision variable is not of type `Enum`.

[18] This is in transition to the new IVML term "annotation". We will keep "attribute" for a certain time and then replace it completely by "annotation".

An `IvmlElement` can automatically be converted to a `String` containing the name of the `IvmlElement`.

3.4.5.2 EnumValue

This subtype of `IvmlElement` represents an IVML enumeration value. However, it redefines some of the inherited operations as follows

- **Any getValue () / Any value ()**
  Returns the enum value itself, i.e., *operand*.
- **String getStringValue () / String stringValue ()**
  Returns the name of the underlying IVML enum literal of the *operand*.
- **Integer getIntegerValue () / Integer integerValue ()**
  Returns the (IVML) ordinal of the *operand*.
- **Real getRealValue () / Real realValue ()**
  Returns the (IVML) ordinal of the *operand* as a `Real`.
- **EnumValue getEnumValue () / Enum enumValue ()**
  Returns the enum value itself, i.e., *operand*.

Due to missing semantics, the operations **getBooleanValue () / Boolean booleanValue ()** are not available on `EnumValue`. Further, `EnumValue` defines the following operations:

- **Integer getOrdinal () / Integer ordinal ()**
  Returns the (IVML) ordinal of the *operand*.

3.4.5.3 DecisionVariable

This subtype of `IvmlElement` represents a configured IVML decision variable. Decision variables that have not been configured / frozen are not accessible and containing expressions cannot be evaluated (will be ignored).

- **sequenceOf(DecisionVariable) variables()**
  Returns the frozen nested variables of *operand*. Except for the IVML types container and compound, this sequence will always be empty. The result cannot be modified.
- **DecisionVariable getByName(String name) / DecisionVariable byName(String name)**
  Returns the decision variable in *operand* (if it exists).
- **IvmlDeclaration getDeclaration() / IvmlDeclaration declaration()**
  Returns the declaration of the decision variable in *operand*.
- **String getVarName() / String varName()**
  Returns the name of the underlying (dereferenced) decision variable of the *operand*. In case of a reference, getName()/name() will return the name of the reference variable but not of the referenced variable.
- **sequenceOf(Annotation) annotations() / attributes()**
  Returns the frozen annotations of *operand*. Except for the IVML types container and compound, this sequence will always be empty. The result cannot be modified.
- **setValue(Object v)**

Changes the value of the underlying decision variable of the *operand* by setting the new value *v*. Attempts to change a frozen variable or to set an incompatible value will lead to a runtime error.

- **Boolean isFrozen()**
  Returns whether the underlying (dereferenced) decision variable of the *operand* is frozen and cannot be changed. In VIL, all variables are frozen while in rt-VIL also unfrozen variables are available.
- **Boolean isConfigured()**
  Returns whether the underlying (dereferenced) decision variable of the *operand* is configured. Please note that `isNull()` implies `isConfigured()`, as the null of IVML configures a decision variable.
- **Boolean isValid()**
  Returns whether the underlying decision variable of the *operand* is valid, i.e., all constraints are fulfilled. Under VIL, this operation always returns `true`. Under rt-VIL the implementation is supposed to define the result[19].
- **DecisionVariable getParent() / parent()**
  Returns the parent variable of *operand* if available. If operand is a top-level variable, i.e., the actual parent is a configuration, the result is undefined.
- **Configuration selectAll()**
  Returns a configuration as a projection of the nested decision variables in *operand* also returned by `variables()` but in terms of a configuration.

A `DecisionVariable` can automatically be converted into `Integer`, `Real`, `Boolean`, `String` or `EnumValue` in terms of its respective value and type. Further, a `DecisionVariable` can automatically be converted into a sequence of decision variables using the `variables()` operation in order to simplify loop declarations.

3.4.5.4 Annotation[20]

This subtype of `IvmlElement` represents a configured IVML annotation. This subtype does not specify any additional operations over `DecisionVariable`.

An `Annotation` can automatically be converted into `Integer`, `Real`, `Boolean`, `String` or `EnumValue` in terms of its respective value and type. Further, an `Annotation` can automatically be converted into a sequence of decision variables using the `variables()` operation in order to simplify loop declarations.

3.4.5.5 IvmlDeclaration

This subtype of `IvmlElement` represents the type underlying an IVML decision variable. Instances of this type do not provide any configuration values rather than providing access to the structure of the represented type, e.g., the nested variable declarations in an IVML compound.

[19] For rt-VIL, the functionality is currently not implemented, but in transition from domain-specific code.

[20] This is in transition to the new IVML term “annotation”. We will keep “attribute” for a certain time and then replace it completely by “annotation”.

3.4.5.6 Configuration

The `Configuration` type provides access to configuration values as well as to the type declarations for a certain IVML model. In particular, the configuration type allows to create projections of a configuration, e.g., to select a subset of decision variables and specify further operations on that subset such as passing it to rules or to (one or more) instantiators. Although being an `IvmlElement`, a configuration instance will not provide access to values.

In traditional product line instantiations, the Configuration will contain *frozen* variables only, i.e., variables that are explicitly intended for instantiation. As VIL is also intended to support runtime variability, i.e., instantiation at runtime in the sense of Dynamic Software Product Lines (DSPL) [10, 11], Configurations may be re-reasoned or changed. We will list specific operations intended for DSPL instantiations in a separate operation list below in this section.

- **sequenceOf(DecisionVariable) variables()**
  Returns the configured and frozen decision variables of *operand*.
- **sequenceOf(Annotation) annotations() / attributes()**
  Returns the configured and frozen annotations of *operand*.
- **Boolean isEmpty()**
  Returns whether configuration in *operand* is empty, i.e., does not contain decision variables. This may occur due to the projection capabilities of this type.
- **DecisionVariable getByName(String n) / DecisionVariable byName(String n)**
  Returns the specified decision variable (if it exists).
- **Configuration selectByName(String n)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are of the type specified in terms of a Java regular expression *n* applied on (qualified and unqualified) decision variable names.
- **Configuration selectByType(String t)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are of the type specified in terms of a Java regular expression *t* applied on (qualified and unqualified) type names.
- **Configuration selectByAnnotation(String n) / selectByAttribute(String n)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are annotated by the annotation specified in terms of a Java regular expression *n* applied on annotation names.
- **Configuration selectByAnnotation(String n, Any v) / selectByAttribute(String n, Any v)**
  Returns a configuration as a projection of *operand* containing those decision variables which are annotated by the specified annotation (in terms of an IVML identifier *n*) and value *v*.
- **Configuration selectByProject(String n)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are defined by the project given by its name *n* and its imported projects. If the project cannot be found, an empty configuration is returned.

- **FileArtifact store(Path p)**
  Stores the underlying (unprojected) configuration of *operand* into *p* containing user-defined configuration values only.
- **FileArtifact store(Path p, Boolean u)**
  Stores the underlying (unprojected) configuration of *operand* into *p* containing user-defined configuration values (if *u* is true) or all configuration values including automatically derived ones (if *u* is false).
- **Configuration selectByProject(String n, Boolean i)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are defined by the project given by its name *n* and, depending whether *i* is true, its imported projects. If the project cannot be found, an empty configuration is returned.

The functions intended to support runtime instantiation of variabilities will be discussed along with rt-VIL in Section 3.7.1.4.

- **Configuration copy()**
  Copies the configuration in *operand*. The variables in the result of this operation can be modified independently from *operand*.
- **Configuration selectFrozen()**
  Projects *operand* to a configuration of frozen variables. A configuration that has been frozen once will remain frozen. This operation is intended to make pre-runtime instantiations explicit.
- **Configuration selectAll()**
  Projects *operand* to all variables (identity projection, i.e., including unfrozen ones). This operation is intended to distinguish between runtime and pre-runtime instantiation scripts. Please note, that a configuration that has been projected once to frozen variables will remain frozen.
- **Configuration reason()**
  Performs a re-reasoning of the *operand*, preferably by applying incremental reasoning techniques. Please check the result of `isValid()` before using the result configuration for instantiation.
- **Boolean isValid()**
  Returns whether the configuration in *operand* is valid and may be considered for instantiation. In particular, performing reasoning on a configuration may result in an invalid configuration.

3.4.6 Built-in Artifact Types and Artifact-related Types

In this section, we will discuss the built-in artifact types as well as artifact-related types such as paths. Please note that the (meta model) of the artifact model is extensible, so that further as well as derived types may be added if needed.

3.4.6.1 Path

A path represents a relative or absolute file or folder. Paths are always relative to the containing project, in more detail to the containing artifact model and normalized in Unix notation (using the slash as path separator). Further, paths may be patterns and contain wildcards according to the ANT conventions [9].

- **String getPath() / String path()**
  Returns a string representation of *operand*.
- **Boolean isPattern()**
  Returns whether the *operand* is a pattern, i.e., whether it contains wildcards.
- **JavaPath toJavaPath()**
  Explicitly converts the *operand* into a Java package path.
- **JavaPath toJavaPath(String p)**
  Explicitly converts the *operand* into a Java package path and removes the prefix Java regular expression *p* (might be due to src folders in Eclipse or Maven).
- **String toOSPath()**
  Turns the *operand* into a relative operating system specific path.
- **String toOSAbsolutePath()**
  Turns the *operand* into an absolute operating system specific path.
- **deleteAll()**
  Deletes all artifacts in the *operand* path. Use this function to delete pattern paths.
- **delete()**
  Deletes the artifact underlying the *operand*. This operation will cause no effects on pattern paths.
- **mkdir()**
  Creates a directory from the *operand* path. This operation will fail if applied to a pattern.
- **setOf(FileSystemArtifact) copy (FileSystemArtifact t)**
  Copies all file system artifacts denoted by the *operand* to the location of *t* and returns the created artifacts at the target location.
- **setOf(FileSystemArtifact) move (FileSystemArtifact t)**
  Moves all file system artifacts denoted by the *operand* o the location of *t* and returns the created artifacts at the target location.
- **Boolean matches(Path p)**
  Returns whether the (pattern in) *operand* matches the given path.
- **setOf(FileArtifact) selectByType(Type t)**
  Returns those artifacts matching *operand* and complying to the given artifact type *t*.
- **setOf(FileArtifact) selectAll()**
  Returns all artifacts matching *operand*.
- **String +(String s)**
  Returns the string concatenation of *operand* and the given String *s*.
- **Boolean exists()**
  Returns whether the artifact denoted by the path exists. The operation will always return false in case of a pattern path.
- **String getName() / String name()**
  Returns the name of the file part of the path or, in case of a pattern path, the entire pattern path.
- **Path rename(String n)**

Renames the *operand* to *n* and returns the related new path. An unqualified name *n* is interpreted in the context of the respective artifact, e.g., for a file artifact the own path is prepended. In general, qualified names are encouraged if possible.

Paths can be constructed from a `String` using the explicit constructor.

Conversions: A `Path` can be converted automatically into a `FolderArtifact`.

3.4.6.2 JavaPath

A subtype of `Path` representing Java package paths (separated by ".").

- **String getPathSegments() / String pathSegments()**
  The path segments of *operand* without the file name.
- **JavaPath getPathSegmentsPath() / String pathSegmentsPath()**
  The path segments of *operand* without the file name.

Conversions: A `String` can be converted automatically into a `JavaPath`.

3.4.6.3 ProjectSettings

The type `ProjectSettings` is an internal super type of all keys that can be used to specify global project settings. Currently, only the Java language extension (Section 3.5.2) defines a key for the classpath.

3.4.6.4 Project

The project type encapsulates the physical location of a product line including all artifacts, in particular in terms of an Eclipse project. There are no explicit constructors for this type, as instances will be provided by the VIL/EASy-Producer runtime environment.

- **String getName() / String name()**
  Returns the name of the project in *operand*.
- **String getPlainName() / String plainName()**
  The name of *operand* without file name extension.
- **setOf(FileArtifact) selectAllFiles()**
  Returns all file artifacts in *operand*.
- **setOf(FileArtifact) selectAllFolders()**
  Returns all folder artifacts in *operand*.
- **setOf(Project) predecessors()**
  Returns all predecessor projects in *operand*. In standalone product lines, the result may be empty. In hierarchical product lines, the predecessors are returned.
- **Path getPath() / Path path()**
  Returns the (base) path the *operand* is located in.
- **Path localize(Project s, Path p)**
  Localizes path *p* originally in project s to the project in *operand*.
- **Path localize(Project s, FileSystemArtifact a)**
  Localizes path of *a* originally in project s to the project in *operand*. This operation does neither copy nor move *a*.

- **setOf(FileArtifact) selectByType(Type t)**
  Returns those file artifacts in the *operand* project which comply with the given type *t*.
- **setOf(FileArtifact) selectByName(String r)**
  Returns those file artifacts in the *operand* project for which their name complies with the Java regular expression in *r*.
- **Path getEasyFolder() / Path easyFolder()**
  Returns the path to the EASy producer configuration files in *operand*.
- **Path getIvmlFolder() / Path ivmlFolder()**
  Returns the path to the IVML models in *operand* (typically the same as `getEasyFolder()`).
- **Path getVilFolder() / Path vilFolder()**
  Returns the path to the VIL build specification models in *operand*.
- **Path getVtlFolder() /Path vtlFolder()**
  Returns the path to the VTL template models in *operand*.
- **setSettings(ProjectSettings key, Any object)**
  Sets the settings for the artifact model
- **Any getSettings(ProjectSettings key)**
  Returns the settings object for the specified key

Conversions: A `Project` can be converted automatically into its base `Path`.

3.4.6.5 Text

Represents an artifact or a fragment artifact in terms of the underlying text and allows direct manipulations. The manipulations will be written back into the artifact at the end of the lifetime of the related VIL variable. A text representation may be modifiable or read-only depending on the underlying artifact.

- **Boolean isEmpty()**
  Returns whether the text representation in *operand* is empty.
- **String getText() / String text()**
  Returns the textual contained in the text representation in *operand*. Please note that the result is disconnected from the textual representation, i.e., changes made on operand will not be reflected into the result and vice versa.
- **setText(String t)**
  Replaces the contents of the text representation in *operand* by t.
- **Boolean matches(String regEx)**
  Returns whether the specified *regEx* matches the textual representation in *operand*.
- **Text substitute(String regEx, String r)**
  Substitutes all occurrences of *regEx* in *operand* by *r* and returns the modified text.
- **Text replace(String s, String r)**
  Substitutes all occurrences of *s* in *operand* by *r* and returns the modified text. This operation does not consider regular expression matches rather than direct text matches.

- **Text append(String s)**
  Appends *s* to the end of *operand* and returns the modified text.
- **Text prepend(String s)**
  Prepends *s* before the beginning of *operand* and returns the modified text.
- **Text append(Text t)**
  Appends the entire contents of *t* to the end of *operand* and returns the modified text.
- **Text prepend(Text s)**
  Prepends the entire contents of *t* before the beginning of *operand* and returns the modified text.

Conversions: Please note that a text representation cannot be converted automatically into a String. This shall avoid confusion as, in contrast to a text representation, the resulting String is disconnected from the underlying artifact and changes to the String will not affect the artifact.

3.4.6.6 Binary

Represents an artifact or a fragment artifact in terms of the underlying binary form and allows direct manipulations. This type is subject to future extensions. The manipulations will be written back into the artifact at the end of the lifetime of the related VIL variable. A text representation may be modifiable or read-only depending on the underlying artifact.

- **Boolean isEmpty ()**
  Returns whether the binary representation in *operand* is empty.

3.4.6.7 Artifact

This is the most common artifact type. All specific artifact types are subtypes of `Artifact`.

- **delete()**
  Delete the artifact in *operand* regardless of whether it is a file, a component, or a fragment within an artifact.
- **Boolean exists ()**
  Returns whether the artifact in *operand* actually exists. Please note that the semantics of this operation depends on the type of artifact, e.g., for a FileArtifact this operation returns whether there is an actual underlying file.
- **String getName () / String name()**
  Returns the name of the artifact in *operand*. The specific meaning of the name depends on the actual artifact type.
- **Text getText() / Text text()**
  Returns the textual representation of the artifact in *operand*. Whether the representation can be manipulated depends on whether the artifact itself may be modified.
- **Binary getBinary() / Binary binary()**

Returns the binary representation of the artifact in *operand*. Whether the representation can be manipulated depends on whether the artifact itself may be modified.

- **rename(String n)**
  Renames the artifact in *operand*. The specific effect of this operation and whether it may be applied at all depends on the actual artifact type.

3.4.6.8 FileSystemArtifact

Represents the most common type of file system artifacts.

- **Path getPath() / Path path()**
  Returns the path to *operand*.
- **setOf(FileSystemArtifact) move(FileSystemArtifact a)**
  Moves the artifact in *operand* to the location of *a*. If *a* exists, it will be overwritten.
- **setOf(FileSystemArtifact) copy(FileSystemArtifact a)**
  Copies the artifact in *operand* to the location of *a*. If *a* exists, it will be overwritten.

Conversions: A `FileSystemArtifact` can be converted automatically into its `Path`.

3.4.6.9 FolderArtifact

This type represents a folder in the file system and always belongs to a certain artifact model (and typically to a containing `Project`). `FolderArtifact` is a subtype of `FileSystemArtifact`.

- **setOf(FileArtifact) selectAll ()**
  Returns all file system artifacts contained in *operand*.

Conversions: `FolderArtifact` can automatically be converted into a `String` containing the path or into a `Path` denoting the location.

3.4.6.10 FileArtifact

This type represents a file in the file system and always belongs to a certain artifact model (and typically to a containing `Project`). `FileArtifact` is a subtype of `FileSystemArtifact`. Please note that the actual instance in a variable of type `FileArtifact` may belong to a subtype as the creation of artifact takes artifact specific rules into account.

A temporary `FileArtifact` can be constructed using the constructor without arguments. A string (containing a path) as well as a `Path` can be converted automatically into a `FileArtifact`.

A `FileArtifact` is created as the default fallback, i.e., if no more specific artifact matches the underlying real artifact. The artifact creation mechanism may be configured using an external Java properties file, which relates artifact names to file path patterns (overwriting or extending the built-in rules).

3.4.6.11 VtlFileArtifact

The `VtlFileArtifact` type is a subtype of `FileArtifact`. In particular, it helps the VTL template instantiator in dynamic dispatch between other types of file artifacts and actual VTL templates. It does not provide additional operations or conversions over the `FileArtifact`.

A `VtlFileArtifact` is created for all real file artifacts with file extension `vtl`.

3.4.6.12 XmlFileArtifact

The `XmlFileArtifact` is a built-in composite artifact, i.e., if it exists it is either empty or contains a valid XML document. Its content is analysed for known substructures, which are made available for querying and manipulation in terms of fragment artifacts. Please see the warnings regarding text operations below.

- **XmlFileArtifact XmlFileArtifact()**
  Creates a temporary XML file artifact. Please note that the file artifact is initially empty and needs the creation of a root element (see below).
- **XmlElement getRootElement() / XmlElement rootElement()**
  Returns the root element of the XML artifact in *operand*.
- **setOf(XmlElement) selectAll()**
  Returns all XML elements contained in *operand*.
- **setOf(XmlElement) selectChilds()**
  Returns all XML elements contained in the root element of *operand*.
- **setOf(XmlAttribute) selectByName (String r)**
  Returns those XML elements in *operand*, which comply with the given name pattern specified as Java regular expression.
- **setOf(XmlAttribute) selectByPath (String p)**
  Returns those XML elements in *operand*, which comply with the given name path *p*. Here, a path is a hierarchical name separated by slashes denoting the names of nested elements (except for the root element which is implicitly selected). Path elements may be Java regular (sub-)expressions.
- **setOf(XmlAttribute) selectByXPath (String p)**
  Returns those XML elements in *operand*, which comply with the given XPath[21] *p*. As *p* is a String, VIL variable and expression replacements happen before evaluating of the path.
- **XmlElement createRootElement(String n)**
  Creates the root element of the *operand* with name *n* if no root element exists. Otherwise, returns the root element.
- **setOf(XmlAttribute) selectByRegEx (String r)**
  Returns those XML elements in *operand*, which comply regarding their name with the given Java regular expression *r*.

A `XmlFileArtifact` is created for all real file artifacts with file extension `xml`. Automated conversion is supported from `String` and `FileSystemArtifact`.

[21] http://www.w3.org/TR/xpath20/

Please be aware of the fact that appending to a `XmlFileArtifact` via the textual representation (although available) may lead to illegal XML documents, and, as the `XmlFileArtifact` works on valid XML documents only, appending operations may be ignored. Further, appending to an empty `XmlFileArtifact` leads to a default valid XML document that potentially does not include the appended elements. In general, we recommend using the operations of the specific XML types. However, if you would like to rely on the textual representation, please either use valid replacements or change the entire text by setting a valid XML string.

XmlElement

The `XmlElement` is a built-in fragment artifact, which belongs to the `XmlFileArtifact`. In particular, it inherits all operations from `Artifact` such as access to the representations of the contained text or CDATA. Please note that deleting the root element of an XML document is not supported due to technical restrictions. However, modifying the root element or deleting and recreating the entire artifact is possible (see Section 4.3.5).

- **XmlElement XmlElement(XmlElement p, String n)**
  Creates a new XmlElement with name *n* as child of the parent element *p*.
- **XmlElement XmlElement(XmlFileArtifact p, String n)**
  Creates a new XmlElement with name *n* as child of the parent XMLFileArtifact *p*. This is just a convenience constructor which actually executes `XMLElement(p.getRootElement(), n).`
- **setOf(XmlElement) selectAll()**
  Returns all XML elements contained in *operand*.
- **setOf(XmlAttributes) attributes()**
  Returns all XML attributes belonging to *operand*.
- **Text getCdata()**
  Returns the CDATA information of *operand* as a textual representation.
- **XmlAttributes getAttribute(String n)**
  Returns the XML attribute in *operand* with the specified name *n*. Please note that this operation does not fail, if the specified attribute does not exist but rather the subsequent evaluation will stop.
- **XmlAttributes setAttribute(String n, String v)**
  Defines or changes the XML attribute in *operand* with the specified name *n* to the given value *v*.
- **setOf(XmlAttribute) selectByName (String r)**
  Returns those XML elements in *operand*, which comply with the given name pattern specified as Java regular expression.
- **setOf(XmlAttribute) selectByName (String r, Boolean c)**
  Returns those XML elements in *operand*, which comply with the given name pattern specified as Java regular expression. The parameter *c* denotes whether the name comparison shall consider case sensitivity.
- **setOf(XmlAttribute) selectByRegEx (String r)**
  Returns those XML elements in *operand*, which comply regarding their name with the given Java regular expression *r*.
- **setOf(XmlAttribute) selectByPath (String path)**

Returns those XML elements in *operand*, which comply with the given name path. Here, a path is a hierarchical name separated by slashes denoting the names of nested elements (except for the root element which is implicitly selected). Path elements may be Java regular (sub-)expressions.

- **setOf(XmlAttribute) selectByXPath (String p)**
  Returns those XML elements in *operand*, which comply with the given XPath[21] *p*. As *p* is a String, VIL variable and expression replacements happen before evaluating of the path.

XmlAttribute

The `XmlElement` is a built-in fragment artifact, which belongs to the `XmlFileArtifact` and to the fragment artifact `XmlElement`. In particular, it inherits all operations from `Artifact`.

- **XmlAttribute XmlAttribute(XmlElement p, String n, String v, Boolean f)**
  Creates a new XML attribute for in *p* with name *n* and value *v*. In case that an equally named attribute already exists in p, the attribute is overwritten if *f* is true or created anyway (*f* is false).
- **XmlAttribute XmlAttribute(XmlElement p, String n, String v)**
  Creates a new XML attribute for in *p* with name *n* and value *v* by executing `XmlAttribute(p, n, v, true)`.
- **String getValue () / String value()**
  Returns the value of the attribute in *operand*.
- **setValue (String v)**
  Changes the value of the attribute in *operand* to *v*.

3.4.7 Built-in Instantiators

VIL provides also several built-in instantiators, in particular to modify or generate entire artifacts in one step. Basically, instantiators shall be defined using the VIL template language (this actually happens through an instantiator for VIL templates). However, very complex instantiation operations as well as existing (legacy) instantiator operations shall be available to VIL, also as a better integrated alternative to just calling an operating system command. In this section, we will discuss the instantiators shipped with the VIL implementation as well as their specific operations. Please refer to the EASy-Producer developer documentation on how to realize custom instantiators.

3.4.7.1 VIL Template Processor

The VIL template processor is responsible for interpreting and executing VIL template scripts in close collaboration with VIL. It may operate in two different modes depending on the actual arguments, namely executing VIL templates or replacing VIL expressions in a file artifact.

Basically, VTL receive three different parameters, the template, the variability configuration and the target artifact (fragment) to be produced. Thereby, the instantiator itself takes an argument of type `Artifact`, but the dynamic dispatch mechanism allows specifying subtypes in the template parameters or even to have multiple main subtemplates for different artifact types. In addition the VIL template processor may receive an arbitrary number of named arguments specific to the template to be executed.

This instantiator provides three instantiator operations:

- **setOf(FileArtifact) vilTemplateProcessor(String n, Configuration c, Artifact a, …)**
- Parses and analyses the template given by its name *n* (version restriction see Section 3.1.4), executes the template specification using the configuration *c* and replaces the content of *a*. Additional named arguments are passed to the VTL template *t* in order to customize the processing and must comply to the additional template parameters.
- **setOf(FileArtifact) vilTemplateProcessor(VtlFileArtifact t, Configuration c, Artifact a, …)**
  Parses and analyses the template in *t*, executes the template specification using the configuration *c* and replaces the content of *a*. Additional named arguments are passed to the VTL template *t* in order to customize the processing and must comply to the additional template parameters.
- **setOf(FileArtifact) vilTemplateProcessor(FileArtifact t, Configuration c, Artifact a)**
  Searches for VIL variables and expressions in *t* (variables given as `$variableName`, expressions as `${expression}`) and replaces them with their actual value as defined in the configuration *c*. The output replaces the content of *a*.

In addition, similar operations are provided which take a **collection of artifacts** as input. At a glance, storing the output produced by the same template in multiple files might be superfluous but it simplifies the application of the VTL template processor in conjunction with operations which return a collection with one element, such as copying a file (without further utilizing the `projectSingle` operation).

3.4.7.2 ZIP File Instantiator

The ZIP file blackbox instantiator allows packing and unpacking ZIP files.

- **setOf(FileArtifact) zip(Path b, Path a, Path z)**
  Packs the artifacts denoted by the path *a* (possibly a path pattern) into the ZIP file denoted by path *z* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting ZIP file. If *z* does not exist, a new artifact is created. If *z* exists, the contents of *z* is taken over into the result ZIP, i.e., the artifacts in *a* are added to *z*.
- **setOf(FileArtifact) zip(Path b, collectionOf(FileArtifact) a, Path z)**
  Packs the artifacts in *a* into the ZIP file denoted by path *z* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting ZIP file. If *z* does not exist, a new artifact is created. If *z* exists, the contents of *z* is taken over into the result ZIP, i.e., the artifacts in *a* are added to *z*.
- **setOf(FileArtifact) unzip(Path z, Path t)**
  Unpacks artifacts in the ZIP file *z* into the target path *t*.
- **setOf(FileArtifact) unzip(Path z, Path t, String p)**

Unpacks those artifacts in the ZIP file *z* matching the ANT pattern *p* into the target path *t*.

3.5 Default Extensions

In addition to the instantiators and artifact types provided as core (built-in) functionality, there are also some default extensions of VIL/VTL that are usually installed along with EASy-Producer. Depending on the application area, individual installations of EASy-Producer may or may not contain these default extensions and, thus, customize VIL/VTL according to the individual needs or even provide alternative implementations. Table 1 relates the extensions to the implementing bundles. In this Section, we will describe the default extensions shown in Table 1 in terms of their providing types or instantiators, respectively.

Extension	**Section**	**Required Bundle(s)**
Velocity	3.5.1	de.uni-hildesheim.sse.easy.instantiator.velocity
Java	3.5.2	de.uni_hildesheim.sse.easy.instantiator.java[22]
AspectJ	0	de.uni_hildesheim.sse.easy.instantiator.aspectJ
XVCL	3.5.3	de.uni-hildesheim.sse.easy.instantiator.xvcl
ANT	3.5.4	de.uni_hildesheim.sse.easy.instantiator.ant
Maven	3.5.4	de.uni_hildesheim.sse.easy.instantiator.maven[22]

Table 1: Default extensions and providing bundles.

3.5.1 Velocity

The velocity bundle integrates Apache Velocity[23] into VIL/VTL.

Types

This extension does not provide additional types.

Instantiators

The Velocity instantiator [4] allows using one of the basic functionalities of EASy through VIL. It provides instantiator calls for individual templates and collections of templates.

- **setOf(FileArtifact) velocity(FileArtifact t, Configuration c)**
  Instantiates the template in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result in *t*.
- **setOf(FileArtifact) velocity(collectionOf(FileArtifact) t, Configuration c)**
  Instantiates the templates in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result in *t*.
- **setOf(FileArtifact) velocity(Path t, Configuration c, Map m)**

[22] Requires a proper execution environment, i.e., JAVA_HOME set or executing JRE set to JDK in Eclipse.

[23] http://velocity.apache.org/engine/devel/user-guide.html#Velocity_Template_Language_VTL:_An_Introduction

Instantiates the template in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result into *t*. Thereby, the variable names in *c* are replaced for the template processing by the name-name mapping in *m*, e.g., to enable a mapping of variabilities to implementation names on this level.

- **setOf(FileArtifact) velocity(collectionOf(FileArtifact) t, Configuration c, Map m)**
  Instantiates the templates in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result into *t*. Thereby, the variable names in *c* are replaced for the template processing by the name-name mapping in *m*, e.g., to enable a mapping of variabilities to implementation names on this level.

3.5.2 Java

The Java extension for VIL/VTL provides several Java-specific types as well as instantiators.

Global Settings Keys (above Types)

On order to resolve type bindings the classpath has to be set. This can be done with the help of the JavaSettings.CLASSPATH key constant. Right now three types are supported.

- **setOf(Path)**
  Specify the classpath as a set of Path.
- **setOf(String)**
  Specify the classpath as set of String
- **String**
  Specify the classpath as String

Types

Java File Artifact

The `JavaFileArtifact` is a built-in composite artifact and allows querying fragment artifacts as well as and (simple) manipulation of Java source code artifacts. Please note that the `JavaFileArtifact` represents a Java compilation unit. In addition to the file artifact operations, this artifact defines the following operations:

- **setOf(JavaClass) classes()**
  Returns all classes defined by the *operand* artifact.
- **JavaClass getClassByName(String n) / classByName(String n)**
  Returns the class in *operand* with name *n*. Nothing happens if the class cannot be found.
- **JavaClass getDefaultClass() / defaultClass()**
  Returns the java class in *operand* with the name of the artifact. Nothing happens if the class cannot be found.
- **setOf(JavaImport) imports()**
  Returns all imports defined by the *operand* artifact.
- **JavaPackage javaPackage()**

Returns the package declaration defined by the *operand* artifact.

- **setOf(JavaQualifiedName) qualifiedNames()**
  Returns all qualified names by the *operand* artifact.
- **renamePackages(String o, String n)**
  Renames all packages in *operand* from *o* to *n*. Nothing happens, if *o* cannot be found.
- **renamePackages(Map m)**
  Renames all packages in *operand* according to the mapping in *m* whereby the keys denote old names and the values new names. Nothing happens, if old names cannot be found.
- **renameImports(String o, String n)**
  Renames all imports in *operand* from *o* to *n*. Nothing happens if *o* cannot be found.
- **renameImports(Map m)**
  Renames all imports in *operand* according to the mapping in *m* whereby the keys denote old names and the values new names. Nothing happens, if old names cannot be found.
- **renameQualifiedNames(String o, String n)**
  Renames all qualified names in *operand* from *o* to *n*. Nothing happens, if *o* cannot be found.
- **renameQualifiedNames(Map m)**
  Renames all qualified names in *operand* according to the mapping in *m* whereby the keys denote old names and the values new names. Nothing happens, if old names cannot be found.
- **modifyNamespace(String o, String n)**
  Renames all packages, imports and qualified names in *operand* from *o* to *n*. Nothing happens, if *o* cannot be found.
- **modifyNamespace(Map m)**
  Renames all packages, imports and qualified names in *operand* according to the mapping in *m* whereby the keys denote old names and the values new names. Nothing happens, if old names cannot be found.
- **Boolean hasAnnotation (String a, String f, String v)**
  Determines whether one of the classes, methods or fields in the *operand* defines the specified annotation named *a* with field *f* and value *v*. The annotation name *a* may be given as qualified or unqualified name[24]. The field name *f* may be given as empty string (then the default name "value" is used).

A `JavaFileArtifact` is created for all real file artifacts with file extension `java`.

JavaClass

The `JavaClass` is a built-in fragment artifact, which belongs to the `JavaFileArtifact`. It represents a single class within a Java compilation unit. The `JavaClass` defines the following operations:

[24] Please note that we parse Java files but do not resolve them so that qualified names are only available if they are explicitly stated in the source code.

- **setOf(JavaAnnotation) annotations()**
  Returns all annotations defined by the *operand*.
- **setOf(JavaAttribute) attribute()**
  Returns all attributes defined by the *operand*.
- **JavaClass getAttributeByName(String n) / attribteByName(String n)**
  Returns the attribute in *operand* with name *n*. Nothing happens if the attribute cannot be found.
- **setOf(JavaMethod) methods()**
  Returns all methods defined by the *operand*.
- **setOf(JavaMethod) classes()**
  Returns all classes defined by the *operand* artifact.
- **setOf(JavaQualifiedName) qualifiedNames()**
  Returns all qualified names defined by the operand artifact.
- **String getName()**
  Returns the identifier defined by the operand artifact.
- **deleteStatement(ExpressionEvaluator evaluator)**
  Deletes the statements selected by *evaluator* within the methods of *operand*. Right now only JavaCalls can be deleted. For examples, see Section 4.3.7.
- **deleteMethodWithCalls(ExpressionEvaluator evaluator)**
  Deletes the methods matching *evaluator* within *operand* and all java calls pointing to this method. For examples, see Section 4.3.7.
- **deleteMethodWithCalls(ExpressionEvaluator evaluator, Any replacement)**
  Deletes the methods selected by *evaluator* within *operand* and all java calls assigned to this method with a replacement that should be inserted for when a method is called. For examples, see Section 4.3.7.

JavaAnnotation

The `JavaAnnotation` is a built-in fragment artifact, which represents a Java annotation attached to a class, a method or an attribute. The `JavaAnnotation` defines the following operations:

- **String getQualifiedName() / String qualifiedName()**
  Returns the qualified name of the *operand*. **getName() / name()** returns the simple name.
- **setOf(String) fields ()**
  Returns the fields of the *operand*.
- **String getAnnotationValue(String f)**
  Returns the value of the annotation for field *f* of the *operand***.** If the field *f* does not exist, VIL / VTL will ignore the related expression.

JavaMethod

The `JavaMethod` is a built-in fragment artifact, which represents a Java method as part of a class. The `JavaMethod` defines the following operations:

- **String getQualifiedName() / String qualifiedName()**
  Returns the qualified name of the *operand*. **getName() / name()** returns the simple name.

- **deleteStatement(Expression e)**

  Deletes all the statements in *operand* that are selected by the expression *e*. Currently, *e* can be of type JavaCall (see below).
- **setOf(JavaAnnotation) annotations()**
  Returns all annotations defined by the *operand*.

JavaCall

A JavaCall is a built-in temporary fragment artifact which is provided by JavaMethod for iterator expressions. A JavaCall defines the following operations:

- **String getType() / String type()**
  Returns the (simple name of the) type the *operand* applies to.

JavaAttribute

The `JavaAttribute` is a built-in fragment artifact, which represents a Java attribute (field) as part of a class. The `JavaAttribute` defines the following operations:

- **String getQualifiedName() / String qualifiedName()**
  Returns the qualified name of the *operand*. **getName() / name()** returns the simple name.
- **setValue(Any value)**
  Defines the (initial) value of the operand. The value will be emitted as part of the attribute declaration.
- **makeConstant()**
  Turns the *operand* into a "constant" (if it is not already a "constant"), i.e., makes it static final.
- **makeVariable()**
  Turns the *operand* into a "variable" (if it is not already a "variable"), i.e., removes static final.
- **setOf(JavaAnnotation) annotations()**
  Returns all annotations defined by the *operand*.

JavaImport

The `JavaImport` is a build-in fragment artifact, which represents a Java import as part of a class. The `JavaImport` defines the following operations:

- **String getName()**
  Returns the name of the *operand* import declaration.
- String rename(String n)
  Sets the n as name of the import *operand*.

JavaPackage

The `JavaPackage` is a build-in fragment artifact, which represents a Java package as part of a class. The `JavaPackage` defines the following operations:

- **String getName()**
  Returns the name of the package declaration in *operand*.

- **String rename(String n)**
  Sets the name of the package in *operand* to *n*.

JavaQualifiedName

The `JavaQualifiedName` is a build-in fragment artifact, which represents a Java package as part of a class. The `JavaQualifiedName` defines the following operations:

- **String getName()**
  Returns the name of the qualified name in *operand*.
- **String rename(String name)**
  Sets the qualified name of *operand*.

Instantiators
Java Compiler

The Java compiler blackbox instantiator allows to directly compile[25] Java source code artifacts from VIL. It provides the following instantiator call

- **setOf(FileArtifact) javac(Path s, Path t, ...)**
  Compiles the artifacts denoted by the source path *s* (possibly a path pattern) into the target path *t*. Additional parameters of the Java compiler can directly be given as named attributes, such as a collection of Strings or Paths or Artifacts denoting the classpath, for example

  ```
  sequenceOf(String) cp = {"$source/lib/myLib.jar"};
  Javac("$source/**/*.java", "$target/bin", classpath=cp);
  ```

- **setOf(FileArtifact) javac(collectionOf(FileArtifact) s, Path t, ...)**
  Compiles the artifacts denoted by *s* into the target path *t*. Additional parameters of the Java compiler can directly be given as named attributes, such as a collection of Strings or Paths or Artifacts denoting the classpath.

JAR File Instantiator

The JAR (Java Archive) file blackbox instantiator allows packing and unpacking JAR files.

- **setOf(FileArtifact) jar(Path b, Path a, Path j)**
  Packs the artifacts denoted by the path *a* (possibly a path pattern) into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting JAR file. If *j* does not exist, a new artifact is created with an empty manifest. If *j* exists, the contents of *j* (including the manifest) is taken over into the result JAR, i.e., the artifacts in *a* are added to *j*.
- **setOf(FileArtifact) jar(Path b, collectionOf(FileArtifact) a, Path j)**
  Packs the artifacts in *a* into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting JAR file. If *j* does not exist, a new artifact is created with an

[25] EASy-Producer must be executed within a JDK so that Java has access to it's internal compiler. A JRE is not sufficient!

empty manifest. If *j* exists, the contents of *j* (including the manifest) is taken over into the result JAR, i.e., the artifacts in *a* are added to *j*.

- **setOf(FileArtifact) jar(Path b, Path a, Path j, Path m)**
  Packs the artifacts denoted by the path *a* (possibly a path pattern) into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting JAR file. If *j* does not exist, a new artifact is created and *m* will be the manifest. If *j* exists, the contents of *j* is taken over into the result JAR, i.e., the artifacts in *a* are added to *j* while the manifest *m* will take precedence over the existing manifest.
- **setOf(FileArtifact) jar(Path b, collectionOf(FileArtifact) a, Path j, Path m)**
  Packs the artifacts in *a* into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting JAR file. If *j* does not exist, a new artifact is created. If *j* exists, the contents of *j* is taken over into the result JAR, i.e., the artifacts in *a* are added to *j* while the manifest *m* will take precedence over the existing manifest.
- **setOf(FileArtifact) unjar(Path j, Path t)**
  Unpacks artifacts in the JAR file *j* into the target path *t*.
- **setOf(FileArtifact) unjar(Path j, Path t, String p)**
  Unpacks those artifacts in the JAR file *j* matching the ANT pattern *p* into the target path *t*.
- **setOf(FileArtifact) unjar(Path j, Path t, Boolean m)**
  Unpacks artifacts in the JAR file *j* into the target path *t* whereby *m* determines whether the manifest shall also be unpacked.
- **setOf(FileArtifact) unjar(Path j, Path t, String p, Boolean m)**
  Unpacks those artifacts in the JAR file j matching the ANT pattern p into the target path t whereby m determines whether the manifest shall also be unpacked.

AspectJ

The AspectJ [1] extension allows to directly compile[25] Java and AspectJ source artifacts from VIL.

Types

This extension does not provide additional types.

Instantiator

The AspectJ extension provides the following instantiator calls

- **setOf(FileArtifact) aspectJ(Path s, Path t, ...)**
  Compiles the artifacts denoted by the source path *s* (possibly a path pattern) into the target path *t*. If *s* is a folder, then all Java and aspect files will be compiled. Additional parameters of the AspectJ compiler can directly be given as named attributes as described above for the Java compiler blackbox instantiator.
- **setOf(FileArtifact) aspectJ(collectionOf(FileArtifact) s, Path t, ...)**

Compiles the artifacts denoted by *s* into the target path *t*. Additional parameters of the AspectJ compiler can directly be given as named attributes as described above for the Java compiler blackbox instantiator.

3.5.3 XVCL

The XVCL extension allows the usage of the XML-based Variant Configuration Language (XVCL)[26] in VIL scripts. However, this is the integration of a 3rd party research prototype, which is not longer maintained. This instantiator is only for demonstration purpose and contains some known bugs (see below).

Types

This extension does not provide additional types.

Instantiators

The XVCL instantiator integrates XVCL into VIL/VTL. Two different kinds of artifacts are needed to use XVCL, which are usually saved as *.xvcl files:

1) XVCL artifacts describe the instantiation of (code) artifacts, based on XML.
2) A specification, which serves as starting point for the instantiation. This file contains the information about source and destination files, the value settings for the transformation, and a list of files which shall be used for the transformation. We suggest the usage of a template file (cf. Sections 3.2 and 3.4.7.1) for the creation of a product specific specification file.

- **xvcl(FileArtifact s)**
  Uses the *s* for instantiation files with XVCL. This file must be structured as follows:

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "null">
<x-frame name="<name of the whole process (will not be used)>"
    outdir="<output destination folder>">
  <set var="<xvcl variable name>" value="<xvcl variable value>" />
  <!-- further variable settings -->
  <set var="dir" value="<sources dir of xvcl files>"/>
  <set var="dtd" value="null"/>
  <set var="out" value="<output destination folder>"/>
  <adapt x-frame="?@dir?<relative path of first file to instantiate>"
    outfile="<destination>"/>
  <!-- further files to instantiate -->
</x-frame>
```

Example

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "null">
<x-frame name="project_foo" outdir="C:/Workspace/Project_target/src">
  <set var="boolVar" value="true" />
  <set var="stringVar" value="bar" />
  <set var="intVar" value="42" />
```

[26] http://xvcl.comp.nus.edu.sg/cms/

```
  <set var="emptySetVar" value="{}" />
  <set var="dir" value=" C:/Workspace/Project_source/xvcl_sources"/>
  <set var="dtd" value="null"/>
  <set var="out" value=" C:/Workspace/Project_target/src"/>
  <adapt x-frame="?@dir?\main\Main.xvcl" outfile="
    C:/Workspace/Project_target/src/main/Main.java"/>
</x-frame>
```

Known Bugs

- After the specification file was passed to XVCL, it is not possible to delete these file via Eclipse/EASy. It seems that XVCL does not release this resource. Hence, Eclipse/EASy must be closed before this file can be deleted or overwritten.
- XVCL is calling `System.exit(1)` in some cases. We try to avoid such circumstances, but we cannot guarantee that we covered all possible cases. In such a situation, XVCL would close the whole Java Virtual Machine, which will also stop the execution of Eclipse/EASy.

3.5.4 ANT / Make

The ANT extension allows the execution of ANT build processes[27] (version 1.9.4) as well as Make scripts[28] (through ANT) from VIL.

Types

This extension does not provide additional types.

Instantiators

- **setOf(FileArtifact) ant(Path r, String b, String t)**
  Executes ANT on the build file *b* within path *r* with target *t*.
- **setOf(FileArtifact) make(Path r, String m, String t)**
  Executes MAKE on the makefile *m* within path *r* with target *t*. This instantiator takes arbitrary named arguments that are passed to MAKE.
  This instantiator needs that MAKE is installed on the executing computer.

3.5.5 Maven

The Maven extension allows the usage of Maven 3.2.3 build processes in VIL scripts.

Types

This extension does not provide additional types[29].

Instantiators

- **maven(Path r)**

[27] The execution of the Java compiler as part of an ANT build-target requires the "fork"-parameter to be defined as "true", e.g. "<javac fork="true" [...]". If this parameter is not set, the Ant blackbox instantiator may complain about "JAVA_HOME" not being set to the JDK-directory even if this variable is set correctly.

[28] This instantiator is still in development, in particular regarding the installation requirements for non-Unix systems.

[29] Currently, this extension does not support Maven parameters. This will follow in a future version.

Executes maven with the pom.xml file in *r* with clean and install targets.

- **maven(Path r, Boolean u)**
  Executes maven with the pom.xml file in *r* with clean and install targets and updates snapshots according to *u*.
- **maven(Path r, String b)**
  Executes maven with the pom.xml file in *b* within *r* with clean and install targets.
- **maven(Path r, String b, Boolean u)**
  Executes maven with the pom.xml file in *b* within r updating snapshots according to *u* with clean and install targets.
- **maven(Path r, sequenceOf(String) t)**
  Executes maven with the pom.xml file in *r* with given targets *t*.
- **maven(Path r, Boolean u, sequenceOf(String) t)**
  Executes maven with the pom.xml file in *r* updating snapshots according to *u* with given targets *t*.
- **maven(Path r, String b, sequenceOf(String) t)**
  Executes maven with the pom.xml file in *b* within *r* with given targets *t*.
- **maven(Path r, String b, Boolean u, sequenceOf(String) t)**
  Executes maven with the pom.xml file in *b* within *r* updating snapshots according to *u* with given targets *t*

Please note that further maven command line arguments may be passed in through the target list *t*. The VIL maven integration supports two Java system properties, namely

- `easy.maven.asProcess`, a Boolean value (`true`|`false`) indicating whether Maven shall be executed in a separate process or directly within the EASy producer process. While executing Maven within EASy producer is easier as it is just a method call, the process variant is advisable as Maven otherwise keeps a hold on certain created files (file descriptor leaks described in the Maven issue tracker).
- `easy.maven.home`, the optional home folder of a local Maven installation (not set by default). Although EASy carries all necessary binaries for maven, this setting can be helpful in standalone execution of EASy-Producer in order to simplify the installation. This implies `easy.maven.asProcess=true`.

3.6 Runtime Variability Instantiation Language

The runtime variability instantiation language (rt-VIL) is an extension of VIL inspired by concepts of Stitch [13] and S/T/A [14] in order to enable runtime instantiation and runtime reconfiguration / adaptation. Basically, rt-VIL follows the core principles DSPLs [10, 11], i.e., it relies on an explicit product line variability model (IVML), which makes runtime-variability available explicit and enables runtime re-configuration / adaptation through monitoring the execution environment and performing adequate runtime changes through re-configuration. Although traditional pre-runtime product line capabilities are not required in DSPLs [10, 11], we explicitly support product line instantiation at runtime through VIL capabilities. As a valid configuration is also required at runtime, we utilize runtime reasoning (in terms of the EASy IVML reasoning support [7]) for detecting invalid configurations and also for value propagation. For enacting runtime changes to the underlying system, we rely on architectural bindings and a translation of the runtime configuration to system-specific concepts. For more details on the approach, in particular in the context of adaptive real-time data stream processing, please refer to [12].

3.6.1 Reserved Keywords

In rt-VIL, the following keywords are reserved and must not be used as identifiers. Please note that this set of reserved keywords is complemented by the keywords of VIL listed in Section 3.1.1 and those defined by the common VIL expression language given in Section 3.3.1.

- **`rtVilScript`** (replacing **`vilScript`** from Section 3.1.1)
- **`breakdown`**
- **`fail`**
- **`objective`**
- **`persistent`**
- **`refail`**
- **`strategy`**
- **`tactic`**
- **`weighting`**

3.6.2 rt-VIL Script

In rt-VIL, a script (`rtVilScript`) is the top-level containing element. This element is mandatory as it identifies the scope for the strategies, tactics and VIL rules to be defined. The definition of a script requires a name in order enable script imports or extensions. rt-VIL scripts can be imported through their name using an import statement. Import statements are given before the script scope. Scripts can be extended akin to classes in object orientation allowing strategies, tactics or rules to be overridden. Further, the definition of a script requires a parameter list specifying the expected information from the execution environment. At least, the source artifact model (carrying the actual state of the runtime components), the actual (runtime) configuration with bound runtime decision variables (from monitoring / analysis), and the target artifact model (containing the instantiated artifacts or components, such as enactment commands) must be provided as parameters.

In a script, all decision variables of the (runtime) configuration are available through their (qualified) name. As IVML configurations may be partial or even composed dynamically, the actual decision variables, their types and type definitions are not necessarily known at the point in time when the script is specified. Thus, the validity of IVML names can only be determined at execution time of the script when also the actual (runtime) configuration is known. This may complicate the development of VIL scripts as actually unknown identifiers will at least lead to a warning. To support the Adaptation Manager in specifying valid and readable scripts, rt-VIL provides the **`advice`** annotation specifying the actual IVML model. This annotation allows using IVML identifiers instead of variable names and, moreover, maps all IVML types into the rt-VIL type system and enables dynamic dispatch also over IVML types.

Syntax:

```
//imports
@advice(ivmlName)
rtVilScript name (parameterList) extends name1 {
  //optional version specification
  //global variable declarations / definitions
  //strategy declarations
  //tactics declarations
  //rule declarations
}
```

Description of syntax:

- First, all referenced scripts are imported. Imports support the modularization of rt-VIL scripts, i.e., a top-level script may import separate scripts handling for example different lifecycle phases such as startup, runtime and shutdown of a pipeline. The syntax for an import statement follows VIL (see Section 3.1.4).
- An optional advice annotation may declare the underlying variability model to map IVML identifiers and types into rt-VIL. The syntax is akin to VIL (see Section 3.1.2). In contrast to VIL, unfrozen fields or attributes can be changed in rt-VIL.
- The keyword `rtVilScript` defines that the identifier name is defined as a new adaptation behavior script with contained strategies, tactics and VIL rules.
- The *`parameterList`* denotes the arguments to be passed to a rt-VIL script for execution. In rt-VIL, the source and target artifact model (given in terms of the type Project), the (runtime) configuration as well as the triggering event are passed in.
- A rt-VIL script may optionally extend an existing (imported) rt-VIL script. This is expressed by the *`extends`* keyword indicating the name of the extended script. The strategies, tactics and rules of the extended script are available, in particular for overriding.
- Further, optional global variables may be declared or defined, respectively. All types defined by the rt-VIL/VIL type system can be used.
- A rt-VIL script may contain strategies, tactics and VIL rules. While there is no predefined sequence of defining strategies and tactics, the actual definition sequence is taken into account if, however, the respective preconditions does not lead to a unique selection of strategies or tactics at execution time. VIL rules can be used to define functions or instantiation tasks. The sequence of VIL rules is not relevant for execution. Extending scripts may override strategies, tactics and rules by specifying the same signature as the original declaration in an extended script. Further, extending scripts may participate in dynamically dispatched execution by defining new typed alternatives.

Example:

```
@advice(QM)
rtVilScript QualiMasterFinancial (Project source,
  Configuration config, Project target,
  AdaptationEvent trigger) {
    // variables, strategies, tactics, etc.
}
```

The example above shows a very basic, empty rt-VIL script. The script is linked against the QualiMaster variability model (just called QM) and called `QualiMasterFinancial`. The script receives the source artifact model, the (runtime) configuration, the target artifact model (that shall be modified) and the actual adaptation event.

3.6.3 Strategy

A strategy represents a high-level adaptation objective capturing the logical aspect of planning an adaptation. It encapsulates the relevant sub-strategies and tactics for the realization of the stated adaptation objective. The impact of executing strategies can be recorded in the QoS database and used to realize proactive adaptation.

Below, we discuss the execution semantics of a rt-VIL script in order to highlight the interplay of strategies, tactics and VIL rules.

- First, the applicable top-level strategies are determined. Multiple strategies may define the same objective so that additional preconditions (including the triggering event type) can be used to differentiate among the applicable strategies. If still multiple strategies are applicable, they are processed in order of their specification. Strategies can handle distinct events, e.g., a startup event to initialize the runtime variables. Initialization of runtime variables such as the active algorithm or functional parameters may also be given in terms of default values in the configuration.
- A strategy can specify multiple sub-strategies and tactics to realize the adaptation towards the actual (failing) objective. The selection of the sub-strategies and tactics during script execution can be static or dynamic. In the static case, the first applicable sub-strategy or tactic in the given sequence is executed depending on their preconditions. In the dynamic case, a weighting function determines the ranking of the sub-strategies and tactics, which, in turn, determines the sub-strategy or tactic to be executed. The weighting function may consider historical information such as the previous impact depending on the functions provided by specific rt-VIL types. Further, we allow sub-strategies to enable a hierarchical breakdown of the adaptation specification along nested structures of the underlying application and, thus, dynamic execution also on the nested levels.
- Ultimately, a tactic is executed to determine the new settings of the runtime configuration. However, a tactic may fail, e.g., as its changes lead to an invalid runtime configuration detected by runtime reasoning. A tactic may revert to the last valid configuration, signal its failure or even cause the end of the

script execution. In case of a failing tactic, the strategy continues with the next applicable sub-strategy or tactic in the sequence of ranking, respectively.

- A strategy succeeds if at least one of its sub-strategies or tactics succeeds. Finally, this determines whether the top-level strategy succeeds or fails.
- If the executed top-level strategy succeeded, the enactment of the runtime configuration starts, i.e., a predefined but overridable VIL-rule (see below) is executed that specifies the mapping from changed decision variables into enactment commands.

Syntax:

As discussed above, a strategy can have an objective, preconditions, or a weighting function and denote sub-strategies and tactics that are supposed to handle the situation indicated by the objective. The syntax of the strategy header follows the syntax of VIL rules.

The body of a strategy is separated into three parts, namely

- the objective (along with supporting local variables),
- the breakdown of the strategy possibly utilizing a weighing function into sub-strategies and tasks and
- post processing.

Actually, the objective is optional, in particular to support startup and shutdown strategies, which may need to be executed regardless of objectives. Also the weighting function is optional in order to enable static (reactive) rule-based adaptation. Please note that both, objective and weighting function may rely on VIL rules or basic operations that can be provided by programmed extensions through the rt-VIL type system. Sub-strategies and tactics can be stated using different syntactical forms as indicated in the syntax below, e.g., including a logical guard expression, a descriptive record (supporting the weighting function) or a maximum execution time. Finally, post processing can execute further VIL rules, e.g., to revert to a previous runtime configuration.

```
strategy name (ParameterList) = post : pre {
  // local variable declarations
  objective expression;
  breakdown {
    weighting (name : expression);
    // optional sub-strategies
    strategy name(ParameterList);
    strategy (guardExpression) name(ParameterList);
    strategy (guardExpression) name(ParameterList)
      with (name = value);
    strategy guardExpression name(ParameterList)
      @numExpression;

    // tactics
    tactic guardExpression name(ParameterList);
  }
  // post processing
}
```

Description of syntax:

- The keyword `strategy` indicates on this level the declaration of a strategy.
- A strategy is identified by its *name*, e.g., for referencing the strategy in other strategies.
- A strategy can declare parameters, which can be used within the strategy or for defining the pre- or post-condition. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Top-level strategies refer to the parameters of the containing script, either specifying all parameters declared by the script (in case of a strategy that performs instantiation) or by just declaring the actual runtime configuration and the triggering event. Sub-strategies may declare specific decision variable types instead of the entire runtime configuration. Further, a strategy may declare the specific event type it reacts on as parameter (and as an implicit precondition).
- Akin to VIL rules, a strategy may have logical pre- and post-conditions. As multiple strategies may define the same objective, the precondition can be used to select among the applicable strategies. If given, the post-condition is checked after the execution of a strategy to determine (in addition to the objective) whether the application of the strategy was successful. If neither pre- nor post-condition are given, also the separating colon can be omitted.
- Within the rule body marked by curly brackets, first local variables can be declared and initialized. In particular, information that is relevant to the calculation of the objective or the weighting function can be collected here, i.e., these variables are intentionally visible to the entire strategy block.
- The objective states a logical condition utilizing observed quality properties (via the runtime variables of the configuration) in expressions such as their target range or utility / cost functions. Please note that the objective is a part of the precondition, which is syntactically highlighted due to its importance for adaptation.
- The breakdown block is executed once to collect the sub-strategies and tactics. This is required to enable the collection of an unknown number of alternatives at design time due to the openness of the configuration in QualiMaster as discussed above. In a second step, the identified sub-strategies and tactics are ranked and executed.
    - The weighting function enables the dynamic selection of the contained sub-strategies and tactics for both, reactive adaptation (without updated common knowledge) and for proactive adaptation (with prediction or updated common knowledge such as QoS impacts). The weighting function dynamically determines the ranking

of the sub-strategies and tactics. Basically it is a function that maps the alternatives (sub-strategies or tactics) under consideration of the actual system state (taken from the runtime configuration or the runtime components) to a real value, e.g., using a utility-cost calculation. Thereby, it acts like an iterator over the alternatives and selects the option with maximum value with access to additional information given in the listing of the sub-strategies and tactics. In particular, rt-VIL operations provide access to the historical impact of a given alternative, the prediction of a certain quality property, or the aggregated quality of a (desired) pipeline. If specified, the weighting function must be given directly at the beginning of the breakdown block.

  - The remainder of the block declares the alternative sub-strategies and tactics. As described above, sub-strategies and tactics can be guarded by an expression, which influences the ranking. Sub-strategies and tactics are referenced by their signature and, in case that a sub-strategy or tactic is used multiple times, a descriptive record (given in terms of name-value pairs) can be given to support the calculation of the weighting function. Please note that the union of all name-value pairs must be consistent over the types implicitly assigned to the names via the individual value expressions used. If no name-value expressions are used, then a Strategy or Tactic instance (see Section 3.7.1) is passed to the weighting function. If at least one name-value expression is used, the implicit record is created and passed to the weighting function. Finally, a maximum execution time (in terms of an integer expression) can be stated, i.e., a relative time bound within the sub-strategy or tactic is either completed or it fails. Timeouts not given as a positive value are ignored.

- The post processing part allows to execute further VIL rules and operations, e.g., to revert to a previous successful runtime configuration. In particular, the post processing part can explicitly cause a strategy to fail (see Section 3.6.5 for the fail statement). Then the next possible strategy is executed. If there is no further strategy, the execution of the script terminates with an adaptation failure.

At the beginning of a strategy, the rt-VIL execution environment calls the (overridable) VIL rule

```
start (Strategy strategy, Configuration config)
```

The built-in implementation opens a transaction for tracking configuration changes. Finally, the execution environment calls the predefined rt-VIL rule

```
Boolean validate (Strategy strategy, Configuration config)
```

at the end of a (so far) successful tactic in order to validate the results of the executed operations. By default, this predefined rt-VIL rule performs runtime reasoning (on the changed variables). This method can be overridden to realize other forms of validation, e.g., for traditional architecture-based adaptation. If the validation or a strategy fails, the execution environment calls

```
        failed (Strategy strategy, Configuration config)
```

to enable rollback or repair operations. By default, a failing strategy leads to a rollback of the previously opened transaction. If the strategy succeeds, the execution environment calls

```
      succeeded(Strategy strategy, Configuration configuration)
```

to enable, which closes the previously opened transaction and commits the changed variables.

Please note that successful top-level strategies call further predefined methods for updating the common knowledge and enactment as we describe in Section 3.6.5.

Example:

The first example below illustrates a simple reactive strategy, which considers exactly one quality parameter, namely the pipeline throughput. As input, the strategy receives the actual QualiMaster runtime configuration (expressed through the type QM made available through a respective advice) and an unspecified adaptation event. In case that a pipeline does not fulfill an overall pipeline throughput value (here, a fictive configuration value specified as a fixed condition in a VIL rule), it aims at changing the pipeline with highest throughput deviation (`candidate`), with a priority on changing a single algorithm parameter. In the extreme case of no successfully executed tactic, the strategy just performs load shedding on the input of `candidate`. Please note that a top-level strategy is implicitly ended by a validation of the changed variables of the runtime configuration and, in case of success, an enactment of the changed variables.

```
strategy infrastructure (QM config, AdaptationEvent trigger) =
{
  setOf(Pipeline) issues = throughputFailingPipelines(config);
  Pipeline candidate = issues->sort(Pipeline p |
     p.capacity).first();

  objective null != candidate;

  breakdown {
    strategy changeSingleParameter(candidate, trigger);
    strategy changeSingleAlgorithm(candidate, trigger);
    // extreme fallback
    tactic shedLoad(candidate, trigger);
   }
}
setOf(Pipeline) througputFailingPipelines(QM config) = {
  config.pipelines-> select(p|
    p.throughput < config.minPipelineThroughput);
}
```

The second example illustrates a strategy with dynamic selection of a comprehensive loop-enumeration of the tactics (based on all alternatives defined in the Configuration). As stated by the signature, the strategy handles a regular adaptation on a `FamilyElement` of a pipeline. The objective of the strategy is that no quality parameter of the family element fails (the related definition is not shown in the example). The weighting function targets the optimization of a utility-cost tradeoff given in terms of two VIL functions (not detailed below). Thereby, additional information defined by the listed tactics is used. For illustration, this example simply enumerates all possible tactics for all available algorithm parameters and family members using map expressions, the VIL version of a loop. Please note that a sub-strategy is implicitly ended by a validation of the changed variables in the runtime configuration.

```
strategy changeAlgorithm (FamilyElement elt,
  RegularAdaptationEvent trigger) = {
  setOf(Quality) failed = failedQualities(elt);
  objective failed.isEmpty();
  breakdown {
    weighting (e: utility(e.elt, e.quality)
      - cost(e.elt, e.quality));
    map(Quality q: failed) {
      map(Parameter p: elt.family.parameter) {
        tactic changeParameter(elt, trigger, p)
          with (elt = p, quality = q);
      }
      map(Algorithm a: elt.family.members) {
        tactic changeAlgorithm(elt, trigger, a)
          with (elt = a, quality = q);
      }
    }
  }
}
```

3.6.4 Tactic

A tactic defines the steps of adaptation to be carried out in order to achieve the objective defined by a calling strategy. A tactic cannot call other tactics or strategies, just rt-VIL operations or rt-VIL rules.

A tactic can have a pre-condition and a post-condition. A tactic is enabled if its precondition holds. A tactic is successful, if its operations are executed successfully and its (optional) post-condition holds. So far, tactics are rather similar to VIL rules. In contrast, tactics serve as a basis for recording the QoS impact through strategies.

Akin to a strategy, a tactic can explicitly fail (see Section 3.6.5 for the fail statement). Then the next possible tactic within the same strategy or the next possible strategy is executed. If there is no further tactic or strategy, respectively, the execution of the script terminates with an adaptation failure.

At the beginning of a tactics, the rt-VIL execution environment calls the (overridable) VIL rule

```
start (Tactic strategy, Configuration config)
```

The built-in implementation opens a transaction for tracking configuration changes. Finally, the execution environment calls the predefined rt-VIL rule

```
Boolean validate (Tactic strategy, Configuration config)
```

at the end of a (so far) successful tactic in order to validate the results of the executed operations. By default, this predefined rt-VIL rule performs runtime reasoning (on the changed variables). This method can be overridden to realize other forms of validation, e.g., for traditional architecture-based adaptation. If the validation or a strategy fails, the execution environment calls

```
failed (Tactic strategy, Configuration config)
```

to enable rollback or repair operations. By default, a failing strategy leads to a rollback of the previously opened transaction. If the strategy succeeds, the execution environment calls

```
succeeded(Tactic strategy, Configuration configuration)
```

to enable, which closes the previously opened transaction and commits the changed variables.

Syntax:

```
tactic name (ParameterList) post : pre {
  // local variable declarations
  // runtime configuration changes, rule calls
  // alternative or iterative execution
}
```

Description of syntax:

- The keyword `tactic` indicates on this level the declaration of a tactic.
- The *name* allows identifying the rule for explicit rule calls or for script extension.
- The *parameterList* specifies explicit parameters which may be used as arguments for precondition rule calls as well as within the rule body. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Parameter must either be bound by the calling strategy.
- The optional post-condition post specifies the expected outcome of the tactic execution in terms of a logic expression.
- The optional precondition pre specifies whether the tactic is considered for execution at all (in addition to the precondition in the strategy). If neither pre- nor post-condition are given, also the separating colon can be omitted.
- The body of the strategy is specified within the following curly brackets. Local variable declarations, rule calls, alternative and looped execution and, in

particular, changes to the runtime variables of the configuration can be specified here.

Example:

This example continues the second example from Section 3.6.3 on changing algorithms and parameters of a QualiMaster pipeline. The tactic shown below performs all operations needed to change a tactic, i.e., it transfers the parameter values of the actual family to the new algorithm and finally changes the current algorithm of the actual family. Please note that a tactic is implicitly ended by a validation of the changed variables in the runtime configuration.

```
tactic changeAlgorithm (FamilyElement elt,
  AlgorithmAdaptationEvent trigger,
  Algorithm newAlgorithm) : {
  Algorithm current = elt.family.current;
  // transfer parameter
  map(Parameter p: elt.family) {
    newAlgorithm.algorithm.parameter
      ->select(q|q.name == p.name).first()
         .setValue(p.value);
  }
  // set algorithm
  elt.family = newAlgorithm;
}
```

3.6.5 Fail / refail statement

Adaptation strategies or tactics may determine that they cannot proceed and that another (less important) strategy or tactic in the same context shall take over. This can be achieved using the `fail` statement. In some cases, e.g., default dynamic dispatch tactics or strategies it is good to cause an already failing to fail again with the original reason. Therefore, rt-VIL offers the `refail` statement. Both statements, `fail` and `refail` can be used within blocks, e.g., alternatives to express a conditional fail.

Syntax:

```
fail "reason"? with intExpression?;
refail;
```

Description of syntax:

- The keyword `fail` indicates that a strategy or tactic shall immediately fail. This statement can be used only within tactics or strategies (post processing part).
- An optional textual reason for failing for documentation purposes. An implementation shall pass this information in a suitable way to the user (as an event, as logging, etc.).
- An optional integer expression indicated by the keyword **`with`** representing an error code as machine-readable form of a reason. An implementation shall pass this information in a suitable way to the user (as an event, as logging,

etc.). If neither an error code nor a failure reason is given, an implementation may use -1 as default error code.

- The keyword `refail` indicates that a strategy or tactic shall immediately fail with the reason of the previous fail. If there was no previous fail, then refail does not have an effect. This statement can be used only within tactics or strategies (post processing part).

Example:

This example is based on the example from Section 3.6.4:

```
tactic changeAlgorithm (FamilyElement elt,
  AlgorithmAdaptationEvent trigger,
  Algorithm newAlgorithm) : {
  Boolean algorithmFound = false;
  // adjust algorithm as shown above
  if (!algorithmFound) {
    fail "no algorithm found" 123;
  }
}
```

In case of a failure, the next viable tactic or strategy will be chosen. If the respective tactic is selected via dynamic dispatch upon another type, the type-specific tactic (the second one below) will not be executed, but the tactic acting as dispatch base (the first one below). In order to cause a failing of the `changeAlgorithm` tactic above to be effective, the typically empty base tactic contains a refail.

```
tactic changeParameter (FamilyElement elt,
  AdaptationEvent trigger) : {
  refail;
}

tactic changeParameter (FamilyElement elt,
  ParameterAdaptationEvent trigger) : {
  // do the parameter adaptation
}
```

3.6.6 Enactment

In runtime reconfiguration based on a variability model, we must validate the constraints of the variability model upon changes of runtime decision variables as explained above. Further, our approach aims to be independent from the underlying system and, thus, needs to specify the mapping from the runtime configuration to the system under adaptation.

In rt-VIL, enactment happens if a top-level strategies succeeds. Then, rt-VIL executes predefined rt-VIL rules (similar to the strategy method), which can be overridden if required. For updating the common adaptation knowledge, e.g., with respect to the impact, rt-VIL calls

```
    update(Strategy strategy, Configuration configuration)
      update(Tactic tactic, Configuration configuration)
```

for each succeeded strategy and tactic. Finally, rt-VIL executes

```
  enact(Project source, Configuration config, Project target)
```

to map and execute the adaptation decisions. The three parameters correspond to the three parameters of the containing script, their types and sequence. Using projection functions of the Configuration type, the runtime configuration can be turned into a subset more appropriate for enactment. Note that there is no predefined enact rule so that it must be defined somewhere in the rt-VIL script.

Example:

An illustrating example combining the enactment of algorithm changes with triggering a subsequent wavefront is shown below as a continuation of the example in Section 3.6.4. Basically, the enact method receives the actual runtime configuration, projects it to the changed variables and considers each changed pipeline and the contained pipeline elements through dynamic dispatch (not all VIL rules for the pipeline types are shown). The enactment of the `FamilyElement` creates the respective commands for the system under adaptation, here in terms of a command sequence. Finally, the enactment rule schedules the subsequent pipeline elements for wavefront adaptation.

```
enact (Project source, Configuration config, Project target) {
  QM changed = config.selectChangedWithContext();
  map(Pipeline p: changed.pipelines) {
    map(Source s: p.sources) {
      enact(p, s);
    }
  }
}
enact(Pipeline pipeline, PipelineElement elt) {
  // dynamic dispatch default - do nothing
}

enact(Pipeline pipeline, FamilyElement elt) {
  Family family = elt.family;
  CommandSequence cmd = new CommandSequence();
  if (null != family.current) { // was changed?
    cmd.add(new AlgorithmChangeCommand(pipeline.name,
      elt.name, family.current.name);
  }
  map (Parameter p: family.parameters) {
    cmd.add(new ParameterChangeCommand(pipeline.name,
      elt.name, p.name, p.value));
  }
  map(Flow f: src) {
    PipelineElement e = f.destination;
    enact(e);
    cmd.add(new ScheduleWavefrontAdaptationCommand(
      pipeline.name, e.name));
  }
  cmd.execute();
}
```

3.6.7 Binding Runtime Variables

So far, we just assumed that certain runtime variables are somehow bound and contain values that reflect the system state at runtime. Actually, binding the values can be done in two ways, namely programmatically and through rt-VIL.

Basically, the underlying adaptive system will perform some form of monitoring to gather the system state and to use this information for adapting itself. As part of monitoring and analysis, the system can also directly bind the respective values in the configuration using the programming API of rt-VIL and IVML. However, in this case, changes to the structure of the variability model also require changes of the binding code and, in particular, programming complex bindings for nested or linked structures can be a complex task.

As a solution, the binding can be directly defined in rt-VIL. Therefore, rt-VIL provides the predefined VIL rule

```
 bindValues (Configuration cfg, mapOf(String, Double) values)
```

which is initially defined empty and can be overridden if required. Thereby, `cfg` is the runtime configuration before adaptation and `values` are the values provided by monitoring (`mapOf(String, Double) values` must now be declared as an additional parameter of the script). The `bindValues` rule is executed before deriving the values of the global parameters, i.e., executing the first strategy. All changes done by bindValues to cfg are not recorded in the configuration change history.

Example

The example below (continuing the previous examples) illustrates the binding of runtime variables using the predefined `bindValues` rule. Actually, the binding is defined in a separate rt-VIL script called mapping to be imported by the main rt-VIL script. As described above, the header declares the specific parameter containing the mapping of monitored values. Actually, it can be rather tedious to work directly on this map. Thus, we assume that the application-specific extension for the underlying system defines a type called `SystemState`, which can be used to wrap the mapping and to access the mapping more conveniently, namely through the observables. An instance of this type is created in the first `bindValues` rule and used in the remainder part of the mapping. Actually, the relevant decision variables are traversed. In this example, the traversal is shown for computing machines (Machine) rather than for pipelines here as this can be done similarly as shown above using dynamic dispatch. The actual values for runtime variables of Machine are assigned in the second `bindValues` rule, which relies on `SystemState` to access the most recent value.

```
@advice(QM)
rtVilScript mapping(Project source, Configuration config,
  Project target, AdaptationEvent event,
  mapOf(String, Real) bindings) {

  // strategies, tactics, enactment

  bindValues(Configuration config,
    mapOf(String, Real) values) = {
    QM qm = config;
    SystemState state = new SystemState(values);
    map(Machine m: qm.machines) {
      bindValues(m, state);
    };
    // go on with relevant parts of the variability model
  }

  bindValues(Machine machine, SystemState state) = {
    machine.bandwidth = state.getMachineObservation(
     machine.name(), ResourceUsage.BANDWIDTH);
  }
}
```

Actually, a rt-VIL script can be executed through the API for just binding values, e.g., to combine monitoring and runtime reasoning over an IVML to detect violated SLAs.

3.7 Built-in types and operations of rt-VIL

In this section we describe the built-in operations of rt-VIL, i.e. types and operations that are specific to rt-VIL and only available in rt-VIL. For describing the types and operations, we follow the conventions of Section 3.4. Basically, all VIL types, operations and instantiators are also available in rt-VIL. Further, we distinguish among built-in types in Section 3.7.1

3.7.1 rt-VIL types

In this section, we detail the operations for the built-in VIL types.

3.7.1.1 RtVilConcept

Represents a the supertype of strategies and tactics to use them, e.g., in weighting functions, update or enactment rule.

- **String getName () / String name ()**
  Returns the name of the *operand* strategy.

No automated conversions or explicit constructors are defined for this type.

3.7.1.2 Strategy

Represents a rt-VIL strategy, e.g., in weighting functions, update or enactment rule. Inherits from `RtVilConcept`.

No automated conversions or explicit constructors are defined for this type.

3.7.1.3 Tactic

Represents a rt-VIL tactic, e.g., in weighting functions, update or enactment rule. Inherits from `RtVilConcept`.

No automated conversions or explicit constructors are defined for this type.

3.7.1.4 ChangeHistory

Collects all changes done to a Configuration (see Sections 3.4.5.6 and 3.7.1.5) in a transition-based way.

- **start()**
  Starts a transaction on *operand*. All changes to variables in the configuration will now be stored in that transaction.
- **rollback()**
  Rolls back the changes in the most recent transaction of *operand*.
- **commit()**
  Committs all changes collected in the most recent transaction of *operand* into the global change set.

3.7.1.5 Configuration

The Configuration type corresponds to the VIL configuration type in Section 3.4.5.6. The functions listed below are intended to support runtime instantiation and may change the configuration.

- **Configuration copy()**
  Copies the configuration in *operand*. The variables in the result of this operation can be modified independently from *operand*.
- **Configuration selectFrozen()**
  Projects *operand* to a configuration of frozen variables. A configuration that has been frozen once will remain frozen. This operation is intended to make pre-runtime instantiations explicit.
- **Configuration selectAll()**
  Projects *operand* to all variables (identity projection, i.e., including unfrozen ones). This operation is intended to distinguish between runtime and pre-runtime instantiation scripts. Please note, that a configuration that has been projected once to frozen variables will remain frozen.
- **Configuration selectChanged ()**
  Projects *operand* to all changed variables.
- **Configuration selectChangedWithContext()**
  Projects *operand* to all changed variables including their context, i.e., their containing variables, their (frozen) variables. Further, the variables referring

to them (including context) and the variables being referenced (including context) are contained in the projection.

- **ChangeHistory changeHistory() / ChangeHistory getChangeHistory()**
  Returns the change history of the *operand*, which is shared among all its projections.
- **Configuration reason()**
  Performs a re-reasoning of the *operand*, preferably by applying incremental reasoning techniques. Please check the result of `isValid()` before using the result configuration for instantiation. Changes through reasoning are tracked in the change history.
- **Boolean isValid()**
  Returns whether the configuration in *operand* is valid and may be considered for instantiation. In particular, performing reasoning on a configuration may result in an invalid configuration.

3.7.2 Types of the underlying adaptive system

Selected types and operations of the underlying adaptive system can be mapped into rt-IVML in order to obtain certain information or to manipulate the system state during enactment. The mapping is system dependent and must be available during specification time (without execution of the operations) and at runtime.

4 How to ...?

Learning a new language is frequently simplified if examples are provided. This is in particular true for languages which include a rich library of operations. In addition to the illustrating examples shown in the sections above, we will discuss a collection of typical application patterns in this section. In particular, this section is meant to be a living document, i.e., this section will be extended over time and is not intended to be comprehensive at the moment.

4.1 VIL

4.1.1 Copy Multiple Files

One recurring task is to copy multiple files, frequently from the source to the target project in order to prepare instantiation. Basically, multiple files can be described by a regular path expression in ANT notation. Let's assume that we want to copy all Java files in the `src` folder of the source project to the `src` folder of the target project:

The hard way

Copying all those files can be described in imperative fashion as follows:

```
Path p = "$source/src/**/*.java";
map(f = p.selectAll()) {
  copy(f, "$target" + p.path().substring("$source".length());
}
```

However, this needs abusing a map as a loop and manually taking care of the target artifact names and it does not care whether files actually need to be copied.

The path copy operation

In order to simplify the copy fragment above, we can just use the related path operation:

```
Path p = "$source/src/**/*.java";
copy(p, $target); // or p.copy($target) if you like
```

Although this is much simpler, it still copies all files.

A bit smarter

Instead of imperative coding, we rely on VIL rules:

```
doCopy() = "$target/src/**/*.java" : "$source/src/**/*.java" {
  copy(FROM, TO);
}
```

In this fragment, the VIL pattern matching algorithm takes care of relating source and target artifacts and copies only files if needed, i.e., the target does not exist or is outdated. However, for each pair of patterns you need a specific rule.

Smart and reusable

To make the copy rule shown above reusable, we introduce a parameter:

```
doCopy(String base) = "$target/$base/**/*.java" :
  "$source/$base/**/*.java" {
  copy(FROM, TO);
}
```

4.1.2 Modifying namespaces

After a copy operation it can occur that some namespaces need modification because the path has changed. Let’s assume we have a package called “test” in our `src` folder and we want to copy it into a new package that contains the name of the project. This modification can be achieved by the following code:

```
postCopy(JavaFileArtifact j, String base) = : {
      j.modifyNamespace("test", "${base}.test");
}


doCopy(String base) = "$target/src/$base/**/*.java" :
  "$source/src/**/*.java" {
  copy(FROM, TO);
  postCopy(TO, base);
}
```

4.1.3 Convenient Shortcuts

Sometimes selection or artifact operations lead to exactly one element as it is intended by the script designer due to the structure of the variability model or the realization of the product line. However, these operations return a collection instead of a single value so that `sequence[0]`, `sequence.get(0)`, `sequence.first()`, `set.asSequence().get(0)` etc. must be applied to obtain that single instance.

In case of sets, this can be simplified by `set.projectSingle()`. Further, if the instance shall further be processed by an instantiator or the VIL template processor, frequently also a collection can be passed in instead of a single artifact so that the operations shown above are not required at all, e.g.,

```
vilTemplateProcessor(”template.vtl”, config, set);
```

4.1.4 Projected Configurations

Frequently, templates or the velocity instantiator do not need access to the full configuration. Basically, passing in subsets of a configuration speeds up the instantiation process. However, in some cases, it is even required to work on a subset of the configuration, e.g., to select a certain element of an IVML container and to continue on the decision variables of that element, e.g., a compound. Consider the following IVML fragment

```
Compound Workflow {
  String name;
  Boolean enabled;
}
sequenceOf(Workflow) workflows;
```

Here, the user may configure different workflows the resulting product shall provide. While generating the workflow implementation (bindings) with VIL or their configuration, typically each configured workflow is processed and the values are taken over. If this shall be realized with velocity, the velocity processor does not know which workflow actually shall be processed. A simple solution in VIL is

```
map(wf = config.selectByName(”workflows”).variables) {
  String wfName = wf.selectByName(”name”).stringValue();
  copy(wfTemplate, ”$target/src/workflows/$wfName.java”);
  velocity(wfFile, wf.selectAll());
}
```

This fragment processes all compound instances in workflows, prepares the instantiation of each workflow by copying the workflow template `wfTemplate` to the right location and by finally by instantiating the template using velocity. Thereby, the nested decision variables from the actual workflow `wf` are projected into a configuration (`wf.selectAll()`) and passed to velocity where then `$name` and `$enabled` can directly be used as placeholders for the actual values.

4.1.5 Running XVCL

As described in Section 3.5.3, XVCL needs a specification file to describe the instantiation of *.xvcl files. The best solution of handling such specification files in EASy is to write a template file. Then use a VIL script to instantiate this template and use the instantiated template for running XVCL. This could be done as below (script for self instantiation):

Template of Specification File

```
template XVCLSpecificationTemplate(Configuration config,
  FileArtifact destFile, Project target) {

  def main(Configuration config, FileArtifact destFile,
    Project target) {
    String codeSourceDir = "${target.getPath()}/xvcl_sources";
    String codeDestination = "${target.getPath()}/src";

    // Template for creating the specification file
    // Header
    '<?xml version="1.0"?>'
    '<!DOCTYPE x-frame SYSTEM "null">'
    '<x-frame name="${destFile.name()}"
      outdir="${codeDestination}">'
    // Product specific value settings
    for (DecisionVariable dv : config.variables()) {
       createVariableAssignment(dv);
    }
    // Footer
    '  <set var="dir" value="${codeSourceDir}"/>'
    '  <set var="dtd" value="null"/>'
    '  <set var="out" value="${codeDestination}"/>'
    '  <adapt x-frame=?@dir?\\main\\Main.xvcl
      outfile="${codeDestination}\\main\\Main.java"/>'
    '</x-frame>'
  }
  def createVariableAssignment(DecisionVariable variable) {
    '  <set var="${variable.name()}"
      value="${variable.getValue()}" />'
  }

}
```

XVCL VIL Script

```
vilScript XVCL_Project (Project source, Configuration config,
  Project target) {

  version v0;

  main(Project source, Configuration config, Project target)
    = : {
    FileArtifact specificationFile =
      "${target.getPath()}/xvcl_sources/0spc.xvcl";
    clean(specificationFile, target);

    vilTemplateProcessor("XVCLSpecificationTemplate", config,
      specificationFile, target=target);
    xvcl(specificationFile);
  }

  clean(FileArtifact specificationFile, Project target) = : {
    specificationFile.delete();
    Path srcPath = "${target.getPath()}/src";
    srcPath.delete();
    srcPath.mkdir();
  }
}
```

4.2 *VIL Template Language*

In this section we will discuss some patterns for the VIL template language.

4.2.1 Don’t fear named parameters

Basically, a VIL template takes two parameters, the configuration and the target artifact. In many situations, already further parameter are helpful, just to parameterize the template or to pass already determined information (from artifacts, the configuration or both) into the template and to simplify the template. Therefore, a VIL template may take an arbitrary number of named parameters.

```
template properties(Configuration config, FileArtifact target,
  String name) {

  def main(Configuration config, FileArtifact target, String
  name) {
   //...
  }
}
```

The respective call in IVML would then look like

```
vilTemplateProcessor (”properties.vtl”, config, target,
  name=”myName”);
```

Please note that this works with arbitrary types and that VIL type conversion applies.

4.2.2 Appending or Prepending

While in some situations the complete creation of an artifact is required, in others it is sufficient to append or prepend information to the contents of an artifact.

The Imperative Style

Basically, we may obtain the contents of the artifact in terms of its textual representation and use the provided operations, e.g.,

```
def main(Configuration config, FileArtifact target) {
  Text targetText = target.getText();
  targetText.append("\n");
  targetText.append("Information ${config.name()}\n");
}
```

The modifications to the text representation will be synched into the artifact as soon as the target variable is reclaimed by the runtime environment, i.e., at the end of the subtemplate. The advantage of this approach is that it works in the same way in the VIL script so that a template may be superfluous. However, stating the required operation in each line and explicitly caring for the line ends is tedious.

Mixing contents

As an alternative, we can simply use the targetText variable within a content statement:

```
def main(Configuration config, FileArtifact target) {
  Text targetText = target.getText();
  '${targetText.text()}

   Information ${config.name()}’
}
```

Please note that a text representation is not automatically converted into a String in order to emphasize that the resulting String is disconnected from the underlying artifact while operations on the text representation will affect the artifact.

4.3 All VIL languages

In this section, we summarize some patterns applicable to both languages (in order to avoid repetitions).

4.3.1 Rely on Automatic Conversions

Retrieving values from a decision variable may become lengthy due to the explicit type access to the actual variable.

```
Boolean value = config.byName(Variability).booleanValue();
```

As a shorter alternative, you may rely on automated conversions, i.e.,

```
Boolean value = config.byName(Variability);
```

4.3.2 Use Dynamic Dispatch

Some datatypes in IVML may be refined, e.g.,

```
compound Element {
// some variables
}
compound RefinedElement refines Element {
}
```

At a glance, processing these datatypes according to their types may lead (in VTL) to expressions such as

```
DecisionVariable elt = …
if (elt.type() == "Element") {
// … process basic elements
} else if (elt.type() == "RefinedElement") {
// … process refined elements
}
```

Actually, using alternatives to make this distinction is similar to object-oriented programming neglecting polymorphism, i.e., using if-cascades with type equality or `instanceof` expressions instead of overridden methods. In VIL and VTL, dynamic dispatch (a more generic form of polymorphism) can be used to achieve an elegant yet extensible specification. While dynamic dispatch is available for all static VIL/VTL types, in particular the artifact types, the IVML types come into play when using the `@advice` annotation (see Section 3.3.11 for details). Assuming that our specification is tagged by an appropriate `@advice` annotation, we can rewrite the specification above (here depicted for VTL but rules in VIL can be stated similary) to

```
def processElement(Element elt) {
// … process basic elements
}
def processElement(RefinedElement elt) {
// … process refined elements
}
```

and replacing the if-cascade by

```
processElement(elt);
```

Moreover, rules or sub-templates in refined scripts (VIL) or templates (VTL) may extend the processing (by adding new rules or sub-templates) or even override existing ones, respectively.

4.3.3 For-loop

Actually, VIL (in terms of `map`) and VTL (in terms of `for`) support iterations only over collections. For iterating over integers, just create a sequence of integers (`createIntegerSequence`) and iterate over that. Please note that this approach currently does not support arbitrary large integer ranges as noted in Section 3.4.3.3. We illustrate this in terms of a VIL fragment below.

```
Integer sum = 0;
sequenceOf(Integer) nums = createIntegerSequence(0, 10);
map(Integer i:nums) {
    sum = sum + i;
};
```

4.3.4 Create XML File / XML elements / XML attributes

XML files can easily be created by defining an `XMLFileArtifact` akin to an ordinary `FileArtifact`. However, initially an `XMLFileArtifact` is empty (except for the xml processing instruction), as no root element is defined. Thus, creating a structurally valid XML file looks like

```
XmlFileArtifact xmlFile = "$target/xml2File.xml";
XmlElement root = xmlFile.createRootElement("MyRoot");
```

Please note that the root element is created on file level, while for putting further elements or attributes into an XML file, the constructor can be used, such as

```
XmlElement hello = new XmlElement(root, "Hello");
new XmlAttribute(hello, "attrib", "value");
```

4.3.5 Overriding / Reinstantiating an XML File

The `XMLFileArtifact` is primarily intended for changing an existing XML file or, as described in Section 4.3.4, for creating new XML files. If an XML file shall be reinstantiated by rewriting a new XML file, you can either

- Delete all nodes in the root element and change the root element accordingy as deleting the root element is not possible due to technical restrictions. Please note that calling `createRootElement` a second time if a valid root exists does not lead to overwriting the root element.
- Delete the existing XML file first and then recreate the root element as needed as shown below.

  ```
  XmlFileArtifact xmlFile = "$target/xml2File.xml";
  xmlFile.delete();
  XmlElement root = xmlFile.createRootElement("MyRoot");
  ```

4.3.6 Print some debugging information

Sometimes it is convenient while developing an instantiation to print out some information, e.g., about variable assignments. Therefore, VIL provides the global `println` operation, which takes any variable value and prints it to the console output. This may look like

```
Integer i = 1;
Integer j = 2;
println("$i $j");
```

4.3.7 How to remove Java calls

Sometimes you may want to remove some java calls (i.e. logging outputs). Let's assume that our application uses the class `EASyLogger` and we want to remove all warning log calls.

Please note that you need to specify the classpath in order for deletions to take place because the type bindings of the related calls need to be resolved properly. Therefore you can set the classpath directly within VIL. Currently, there are three different ways to specify the classpath. You can specify the classpath as string, set of string or set of path as we show below.

Classpath as set of String

```
main(Project source, Configuration conf, Project target) = : {
    setOf(String) classpath = {"$target", "cp", "test"};
    target.setSettings(JavaSettings.CLASSPATH, classpath);
}
```

Classpath as set of Path

```
main(Project source, Configuration conf, Project target) = : {
    Path newClasspath = target.path();
    Path secondClasspath = target.path() + "/thisIsATest";
    setOf(Path) classpath = {newClasspath, secondClasspath};
    target.setSettings(JavaSettings.CLASSPATH, classpath);
}
```

Classpath as String

```
main(Project source, Configuration conf, Project target) = : {
    target.setSettings(JavaSettings.CLASSPATH, "$target");
}
```

Once the classpath is specified you can start modifying your target code. Removing all warning calls as stated above can be archived by the following code.

```
removeWarnings(JavaFileArtifact j) = : {
    JavaClass cls = j.defaultClass();
    cls->deleteStatement(JavaCall c | c.getType() ==
    "EASyLogger" and c.getName() == "warn");
}
```

You can even go further and delete whole methods that you don't want to use. All assigned calls to this method will be removed as well. Let's assume you want to delete a method called "myMethod" which has no return type. This modification can be archived by the following code:

```
removeMethod(JavaFileArtifact f) = : {
    JavaClass cls = f.defaultClass();
    cls->deleteMethodWithCalls(JavaMethod c | c.getName() ==
    "myMethod");
}
```

If you want to delete a method that has a return type you can add a replacement which will inserted at all java calls. If we assume that the method “myMethod” returns a String the code will be the following:

```
removeMethod(JavaFileArtifact f) = : {
    String message = "Method has been deleted";
    JavaClass cls = f.defaultClass();
    cls->deleteMethodWithCalls(JavaMethod c | c.getName() ==
    "myMethod", message);
}
```

5 Implementation Status

The development and realization of VIL and VTL related tools is still in progress. In this section, we summarize the current status.

Missing / incomplete functionality

- Collection of affected artifacts in VIL e.g., through `map` may be incomplete.

6 VIL Grammars

In this section we depict the actual grammar for the VIL languages. The grammar is given in terms of a simplified xText[30] grammar (close to ANTLR[31] or EBNF). Simplified means, that we omitted technical details used in xText to properly generate the underlying EMF model as well as trailing “;” (replaced by empty lines in order to support readability). Please note that some statement-terminating semicolons are optional in order to support various user groups each having individual background in programming languages.

6.1 VIL Grammar

```
ImplementationUnit:
  Import*
  Require*
  LanguageUnit*;

LanguageUnit:
  Advice* 'vilScript' Identifier '(' ParameterList? ')'
  (ScriptParentDecl)? '{'
    VersionStmt?
    LoadProperties*
    ScriptContents
  '}' ';'?

Require:
  'requireVTL' STRING versionSpec ’;’

ScriptParentDecl:
  'extends' Identifier

LoadProperties:
  'load' 'properties' STRING ';'

ScriptContents:
  (VariableDeclaration | TypeDef | RuleDeclaration)*

RuleDeclaration: // Type is for future extension, ignored by now
  (Type? RuleModifier? Identifier '(' (ParameterList)? ')' '=')?
  RuleConditions?
  RuleElementBlock ';'?

RuleConditions:
  (LogicalExpression)?  ':'
  (LogicalExpression (',' LogicalExpression)*)?
```

[30] http://www.eclipse.org/Xtext/

[31] http://www.antlr.org

```
RuleElementBlock:
  '{' RuleElement* '}'

RuleElement:
  VariableDeclaration
  | ExpressionStatement

RuleModifier:
  'protected'

ExpressionStatement:
    ((Identifier ('.' Identifier)? '=')? Expression ';')
    | Alternative ';'?

PrimaryExpression:
  ExpressionOrQualifiedExecution
  | UnqualifiedExecution
  | SuperExecution
  | SystemExecution
  | Map
  | Join
  | Instantiate
  | ConstructorExecution

MapVariable:
  Type? Identifier

Map:
  'map' '(' MapVariable (',' MapVariable)* ('=', ':') Expression ')'
  RuleElementBlock ';'?

Alternative:
    'if' '(' Expression ')' StatementOrBlock
    ('else' StatementOrBlock)?

StatementOrBlock:
    ExpressionStatement | RuleElementBlock

Join:
  'join' '(' JoinVariable ',' JoinVariable ')'
  ('with' '(' Expression ')')?

JoinVariable:
  'exclude'? Identifier ':' Expression
SystemExecution:
  'execute' Call SubCall*

Instantiate:
  'instantiate' (Identifier | STRING)
    '(' ArgumentList? ')' VersionSpec?
```

6.2 VIL Template Language Grammar

```
LanguageUnit:
  Import*
  Extension*
  Advice*
  IndentationHint?
  FormattingHint?
  'template' Identifier '(' ParameterList? ')'
  ('extends' Identifier)? '{'
    VersionStmt?
    TypeDef*
    VariableDeclaration*
    VilDef*
  '}'

IndentationHint:
  '@indent' '(' IndentationHintPart (',' IndentationHintPart)* ')'

IndentationHintPart:
  Identifier '=' NUMBER

FormattingHint:
  '@format' '(' FormattingHintPart (',' FormattingHintPart)* ')'

FormattingHintPart:
    Identifier '=' STRING
;

VilDef:
  'protected'? 'def' Type? Identifier
  '(' ParameterList? ')' StmtBlock ';'?

StmtBlock:
    '{' Stmt* '}'

Stmt:
  VariableDeclaration
  | Alternative
  | Switch
  | StmtBlock
  | Loop
  | ExpressionStatement
  | Content

Alternative:
  'if' '(' Expression ')' Stmt (=> 'else' Stmt)?

Content:
  (STRING) ('|' Expression ';')?

Switch:
  'switch' '(' Expression ')' '{'
    (SwitchPart (',' SwitchPart)* (',' 'default' ':' Expression)?)
  '}'

SwitchPart:
  Expression ':' Expression

Loop:
  'for' '(' Type Identifier ':' Expression
    (',' PrimaryExpression)
      (',' PrimaryExpression)?
     ')'? Stmt
Extension:
  'extension' JavaQualifiedName ';'

JavaQualifiedName:
    Identifier ('.' Identifier)*
```

6.3 *Common Expression Language Grammar*

Actually, parts of this common language are overridden and redefined by the two VIL language grammars.

```
LanguageUnit:
  Import*
  Advice*
  Identifier
  VersionStmt?

VariableDeclaration:
  'const'? Type Identifier ('=' Expression)? ';'

TypeDef:
  'typedef' Identifier Type '; '

Advice:
  '@advice' '(' QualifiedName ')' VersionSpec?

VersionSpec:
  'with' Expression

VersionedId:
  'version' VersionOperator VERSION

VersionOperator:
  '==' |'>' |'<' |'>=' |'<='

ParameterList:
  (Parameter (',' Parameter)*)

Parameter:
  Type Identifier

VersionStmt:
  'version' VERSION ';'
```

```
Import:
  'import' Identifier VersionSpec? ';'

ExpressionStatement:
  (Identifier ('.' Identifier)? '=')? Expression ';'
Expression:
  LogicalExpression | ContainerInitializer

LogicalExpression:
  EqualityExpression  LogicalExpressionPart*

LogicalExpressionPart:
  LogicalOperator EqualityExpression
LogicalOperator:
  'and' |'or' |'xor'

EqualityExpression:
  RelationalExpression EqualityExpressionPart?

EqualityExpressionPart:
  EqualityOperator RelationalExpression

EqualityOperator:
  '==' | '<>' | '!='

RelationalExpression:
  AdditiveExpression RelationalExpressionPart?

RelationalExpressionPart:
  RelationalOperator AdditiveExpression

RelationalOperator:
  '>' | '<' | '>=' | '<='

AdditiveExpression:
  MultiplicativeExpression AdditiveExpressionPart*

AdditiveExpressionPart:
  AdditiveOperator MultiplicativeExpression

AdditiveOperator:
  '+' | '-'

MultiplicativeExpression:
  UnaryExpression MultiplicativeExpressionPart?

MultiplicativeExpressionPart:
  MultiplicativeOperator UnaryExpression

MultiplicativeOperator:
  '*' | '/'

UnaryExpression:
    UnaryOperator? PostfixExpression

UnaryOperator:
  'not' | '!' | '-'

PostfixExpression:  // for extension
  PrimaryExpression

PrimaryExpression:
  ExpressionOrQualifiedExecution
  | UnqualifiedExecution
  | SuperExecution
  | ConstructorExecution
ExpressionOrQualifiedExecution:
  (Constant | '(' Expression ')') SubCall*

UnqualifiedExecution:
  Call SubCall*

SuperExecution:
  'super' '.' Call SubCall*

ConstructorExecution:
  'new' Type '(' ArgumentList? ')' SubCall*

SubCall:
  ('.' | '->') Call | '[' Expression ']'

Declarator:
    Declaration (';' Declaration)* '|'

Declaration:
    Type? DeclarationUnit (',' DeclarationUnit)*

DeclarationUnit:
    Identifier ('=' Expression)?

Call:
  QualifiedPrefix '(' decl=Declarator? param=ArgumentList? ')'

ArgumentList:
  NamedArgument (',' NamedArgument)*

NamedArgument:
  (Identifier '=')? Expression

QualifiedPrefix:
  Identifier ('::' Identifier)*

QualifiedName:
  QualifiedPrefix ('.' Identifier)*

Constant:
  NumValue  |  STRING  |  QualifiedName  |  ('true'  |  'false')  |  null
    | VERSION

NumValue :
  NUMBER

Identifier:
  ID | VERSION | EXPONENT | 'version'

Type:
  QualifiedPrefix
  | ('setOf' TypeParameters)
  | ('sequenceOf' TypeParameters)
  | ('mapOf' TypeParameters)
  | ('callOf' Type? TypeParameters)

TypeParameters:
  '(' Type (',' Type)* ')'

ContainerInitializer:
  '{' (ContainerInitializerExpression
    (',' ContainerInitializerExpression)*)? '}'

ContainerInitializerExpression:
    LogicalExpression |ContainerInitializer

terminal VERSION:
  'v' ('0'..'9')+ ('.' ('0'..'9')+)*

terminal ID:
  ('a'..'z'|'A'..'Z'|'_'|'$') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*

terminal NUMBER:
  '-'?
  (('0'..'9')+ ('.' ('0'..'9')* EXPONENT?)?
  | '.' ('0'..'9')+ EXPONENT?
  | ('0'..'9')+ EXPONENT)

terminal EXPONENT:
  ('e'|'E') ('+'|'-')? ('0'..'9')+

terminal STRING :
  '"'
    ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )*
  '"' | "'"
    ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )*
  "'"

terminal ML_COMMENT:
  '/*' -> '*/'

terminal SL_COMMENT:
  '//' !('\n'|'\r')* ('\r'? '\n')?
```

```
terminal WS:
  (' '|'\t'|'\r'|'\n')+

terminal ANY_OTHER:
```

6.4 rt-VIL Grammar

As discussed in Section 3.6, rt-VIL is an extension of VIL for runtime reconfiguration and adaptation. In this section, we illustrate the grammar of rt-VIL.

```
ImplementationUnit:
  Import*
  Require*
  LanguageUnit*

LanguageUnit:
  (Advice)*
  'rtVilScript' Identifier
  '(' ParameterList? ')'
  (ScriptParentDecl)?
  '{'
    VersionStmt?
    rtContents
  '}' ';'?

rtContents:
  (
    GlobalVariableDeclaration
    | RuleDeclaration
    | StrategyDeclaration
    | TacticDeclaration
  )*

GlobalVariableDeclaration:
  'persistent'?
  VariableDeclaration

StrategyDeclaration:
  'strategy' Identifier
  '(' (ParameterList)? ')'
  '=' RuleConditions?
  '{'
  VariableDeclaration*
  ('objective' Expression ';')?
  ('breakdown' '{'
    weighting=WeightingStatement?
    BreakdownElement+ '}'
  )
  RuleElement*
  '}'
  ';'?

BreakdownElement:
  VariableDeclaration
  | ExpressionStatement
  | BreakdownStatement

WeightingStatement:
    'weighting' '(' name=Identifier ':' expr=Expression ')' ';'

BreakdownStatement:
  ('strategy' | 'tactic')
  (
    '('LogicalExpression ')'
  )?
  QualifiedPrefix
  '(' ArgumentList? ')'
  (
    'with' '('    BreakdownWithPart (',' BreakdownWithPart)')'
  )?
  (
    '@' Expression
  )?
  ';'

BreakdownWithPart:
  Identifier '=' Expression

TacticDeclaration:
  'tactic' Identifier
  '(' (ParameterList)? ')'
  '='RuleConditions?
  RuleElementBlock
  ';'?

RuleElementBlock:
  '{' IntentDeclaration?  RuleElement* '}'

RuleElement:
  VariableDeclaration
  | ExpressionStatement
  | FailStatement

IntentDeclaration: 32
  'intent' ExpressionStatement

FailStatement:
  (('fail' STRING? 'with' Expression?) | 'refail') ';'
```

32 Just syntax for now, intended for detailed adaptation logs about the intent of an adaptation step.

References

[1] Project homepage AspectJ, 2011. Online available at: http://www.eclipse.org/aspectj/.

[2] Eclipse Foundation. Xtend - Modernize Java, 2013. Online available at: http://www.eclipse.org/xtend.

[3] INDENICA Consortium. Deliverable D2.1: Open Variability Modelling Approach for Service Ecosystems. Technical report, 2011. Available online at http://sse.uni-hildesheim.de/indenica

[4] INDENICA Consortium. Deliverable D2.4.1: Variability Engineering Tool (interim). Technical report, 2012. Available online http://sse.uni-hildesheim.de/indenica

[5] INDENICA Consortium. Deliverable D2.2.2: Variability Implementation Techniques for Platforms and Services (final). Technical report, 2013. Available online at http://sse.uni-hildesheim.de/indenica

[6] Object Management Group, Inc. (OMG). Object Constraint Language. Specification v2.00 2006-05-01, Object Management Group, May 2006. Available online at: http://www.omg.org/docs/formal/06-05-01.pdf.

[7] H. Eichelberger, S. El-Sharkawy, C. Kröher, K. Schmid, IVML language specification. http://projects.sse.uni-hildesheim.de/easy/docs/ivml_spec.pdf [validated: February 2015].

[8] Richard M. Stallmann, Roland McGrath, and Paul D. Smith. GNU Make - A Program for Directing Recompilation - GNU make Version 3.82, 2010. Online available at: http://www.gnu.org/software/make/manual/make.pdf.

[9] The Apache Software Foundation. Apache Ant 1.8.2 Manual, 2013. Online available at: http://ant.apache.org/manual/index.html.

[10] S. Hallsteinsen and M. Hinchey and S. Park and K. Schmid, Dynamic Software Product Lines, IEEE Computer 41 (4), pages 93-95, 2008

[11] M. Hinchey, S. Park, and K. Schmid. Building dynamic software product lines. *Computer*, 45(10):22–26, 2012.

[12] QualiMaster Consortium, Deliverable D4.1, Quality-aware Pipeline Modeling, Technical report, 2014, Available online at http://qualimaster.eu

[13] S.-W. Cheng, D. Garlan, B. Schmerl. Stitch: A Language for Architecture-based Self-adaptation. *J. Syst. Softw.*, 85(12):2860–2875, 2012.

[14] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, S. Kounev. Modeling Run-time Adaptation at the System Architecture Level in Dynamic Service-oriented Environments. *Serv. Oriented Comput. Appl.*, 8(1):73–89, March 2014.